

PATTERN DISCOVERY FOR PARALLELISM IN FUNCTIONAL LANGUAGES

Adam David Barwell

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2018

Full metadata for this item is available in
St Andrews Research Repository
at:
<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:
<http://hdl.handle.net/10023/15641>

This item is protected by original copyright

Pattern Discovery for Parallelism in Functional Languages

Adam David Barwell



University
of
St Andrews

This thesis is submitted in partial fulfilment for the degree of
Doctor of Philosophy
at the University of St Andrews

March 2018

Abstract

No longer the preserve of specialist hardware, parallel devices are now ubiquitous. Pattern-based approaches to parallelism, such as *algorithmic skeletons*, simplify traditional low-level approaches by presenting composable high-level patterns of parallelism to the programmer. This allows optimal parallel configurations to be derived automatically, and facilitates the use of different parallel architectures. Moreover, parallel patterns can be swap-replaced for sequential recursion schemes, thus simplifying their introduction. Unfortunately, there is no guarantee that recursion schemes are present in all functional programs. *Automatic pattern discovery* techniques can be used to discover recursion schemes. Current approaches are limited by both the range of analysable functions, and by the range of discoverable patterns. In this thesis, we present an approach based on *program slicing* techniques that facilitates the analysis of a wider range of explicitly recursive functions. We then present an approach using *anti-unification* that expands the range of discoverable patterns. In particular, this approach is *user-extensible*; i.e. patterns developed by the programmer can be discovered without significant effort. We present prototype implementations of both approaches, and evaluate them on a range of examples, including five parallel benchmarks and functions from the Haskell Prelude. We achieve maximum speedups of $32.93\times$ on our 28-core hyperthreaded experimental machine for our parallel benchmarks, demonstrating that our approaches can discover patterns that produce good parallel speedups. Together, the approaches presented in this thesis enable the discovery of more *loci* of potential parallelism in pure functional programs than currently possible. This leads to more possibilities for parallelism, and so more possibilities to take advantage of the potential performance gains that heterogeneous parallel systems present.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Kevin Hammond, for his guidance throughout my studies. His advice and patience has been invaluable for not only developing ideas, but also for providing encouragement.

I would also like to thank Dr Christopher Brown for his ongoing assistance as my second supervisor. His knowledge and perspective on a wide range of topics has never not been a great help.

My special thanks to my fellow PhD students, Chris S., Matus, and Franck, with whom I have had the honour to share an office throughout my studies. Our conversations have always been full of insights, ideas, and information that I would be all the poorer for without. My special thanks also to David, a fellow student under Kevin, without whom the final stretch would have been a lot harder. Thanks to Jan, for his friendship and advice on all things St Andrean. I am also grateful to the other members of the St Andrews Functional Programming group.

Finally, I thank my parents, Frederick and Gillian, for their support and patience throughout my university life.

This work was supported by the EU FP7 grant ‘ParaPhrase:Parallel Patterns for Adaptive Heterogeneous Multicore Systems’ (no. 288570); by the EU H2020 grant ‘RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications – a Software Engineering Approach” (ICT-644235), by COST Action IC1202 (TACLe), supported by COST (European Cooperation in Science and Technology); and by EPSRC grant ‘Discovery: Pattern Discovery and Program Shaping for Manycore Systems’ (EP/P020631/1).

Contents

Contents	ix
1 Introduction	1
1.1 Parallelism	2
1.2 Automatic Pattern Discovery	5
1.3 Contributions	10
1.4 Publications	12
2 Existing Parallelisation Approaches	15
2.1 Automatic Approaches to Parallelisation	16
2.2 Semi-Automatic Approaches to Parallelisation	17
2.3 Summary	24
3 Automatic Pattern Discovery for Parallelisation	25
3.1 Program Calculation Approaches	26
3.2 Non-Calculational Approaches	31
3.3 Summary	33
4 Automatic Pattern Discovery <i>via</i> Program Slicing	35
4.1 Illustrative Example	36
4.2 Preliminaries and Assumptions	42
4.3 Determining Obstructiveness	47
4.4 Implementation	56
4.5 Refactoring to Introduce Map Operations	56

4.6	Examples	64
4.7	Summary and Discussion	93
5	Automatic Pattern Discovery <i>via</i> Anti-Unification	101
5.1	Introduction	102
5.2	Preliminaries and Assumptions	107
5.3	Argument Derivation <i>via</i> Anti-Unification	112
5.4	Deriving Pattern Arguments	121
5.5	Implementation	126
5.6	Finding <code>unfold</code>	128
5.7	Examples	133
5.8	Summary and Discussion	148
6	Conclusions and Future Work	151
6.1	Main Achievements	151
6.2	Future Work	163
6.3	Concluding Remarks	164
A	Proof for Soundness of Slicing Algorithm	165
B	Proof for Soundness of Anti-Unification Algorithm	175
C	Patterns Used in the Anti-Unify Module Refactoring	179
	Bibliography	181

Introduction

The overall goal of this thesis is to develop new approaches to discover *loci* of potential parallelism in pure functional programs in order to facilitate the introduction of parallelism. We introduce two novel approaches to automatic pattern discovery. Parallel programming allows programs to take advantage of the potential performance gains that are offered by parallel hardware. In response to complex low-level approaches, the past few decades have seen the development of a range of techniques that are designed to simplify the parallelisation process. In particular, structured approaches present composable high-level patterns to the programmer that are usually defined as *higher-order functions* in functional languages. While these approaches do not describe how or where parallel patterns can be introduced, *recursion schemes*, which are often also defined as higher-order functions, can be used as *loci* of potential parallelism. The introduction of parallelism then becomes changing a call to a specific recursion scheme instance to a call to the desired equivalent parallel pattern. Unfortunately, there is no guarantee that all recursion scheme instances are used in a program, or that the scheme that most closely reflects the traversal behaviour of the operation is used. An absence of recursion scheme instances, or the use of a more general scheme in place of a valid specialisation, can reduce the opportunities for parallelism, or for alternative parallel structures. *Automatic pattern discovery* techniques can be used to avoid this possibility, by discovering instances of recursion schemes in pure functional programs, and therefore *loci* of potential par-

allelism. Current pattern discovery approaches are often limited both in the range of functions they can analyse, and in the range of patterns that they can discover. This limits the number of recursion scheme instances that can be discovered for a given program, and therefore also opportunities for parallelism. The automatic pattern discovery approaches presented in this thesis build upon existing approaches, and address both of these limitations. We use *program slicing* and *anti-unification* techniques to enable the discovery of patterns in pure functional code. In particular, our anti-unification approach is *user-extensible*, where patterns developed by the programmer can be discovered without requiring significant effort to extend the approach. To the best of our knowledge, this is the first user-extensible approach to automatic pattern discovery. Improving automatic pattern discovery techniques simplifies the parallelisation process by making it increasingly automatic, without sacrificing the range of parallel structures that we can introduce, or the performance gains that they provide. Consequently, this allows programmers to easily take advantage of heterogeneous parallel hardware, and so improve the performance of their programs, with minimal effort or opportunities for error.

1.1 Parallelism

No longer the preserve of servers and supercomputers, parallel processors are now ubiquitous [11]. Multi-core processors can be found in many of the devices with which we surround ourselves; from laptop and tablet computers, mobile telephones, to portable games consoles. This is a trend that is set to continue, as epitomised by Intel's new 18-core i9-7980XE X-series desktop processor. Moreover *heterogeneous* systems, including distributed computing power such as Amazon AWS, and increased prevalence of accelerators, such as FPGAs, discrete GPUs, and Digital Signal Processors, are becoming more common. Consequently, parallel programming is an increasingly necessary skill.

Traditional, predominantly imperative, parallel programming techniques often consist of low-level primitives and libraries that require manual management of threads, communication, locking, synchronisa-

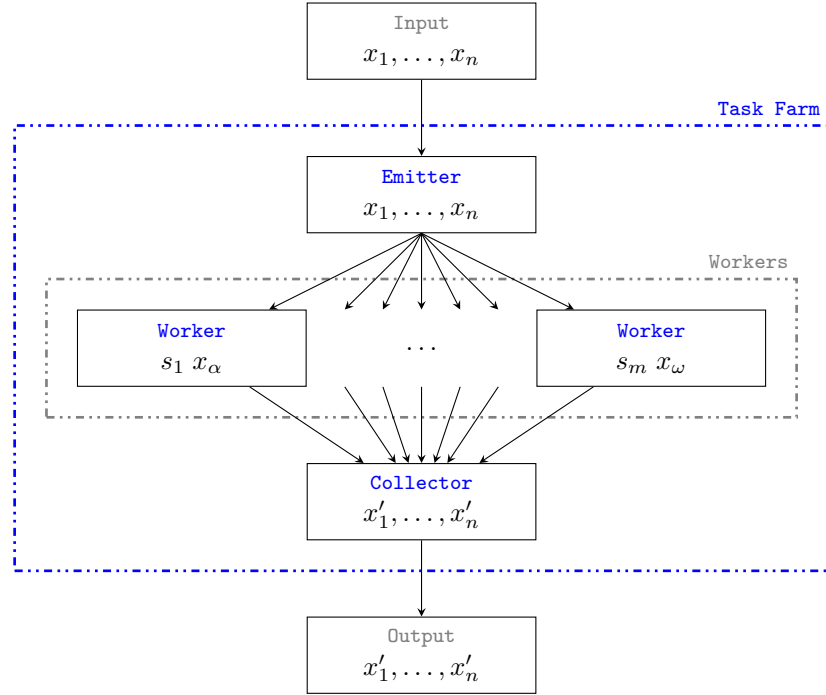


Figure 1.1: A task farm, or parallel map, skeleton. Applies a nested skeleton or function, s_i , to each input, x_j , in parallel.

tion, etc. These low-level components often mean that traditional approaches are tedious, difficult, and error-prone [56]. This makes parallel programming largely inaccessible to the average programmer. More recent structured approaches to parallelism, such as *algorithmic skeletons* [38], simplify the parallelisation process by abstracting away these low-level components. There is a close, and long-observed, correspondence between these structured approaches and functional programming. Futures [110], the Strategies library [111], the Par Monad [87], the Accelerate library [33], the actor model [7], and algorithmic skeletons are all structured approaches for functional languages, where the algorithm and parallel structure of a program are defined separately.

Skeletons, in particular, are implementations of language-independent high-level composable patterns. To introduce parallelism, the programmer calls the skeleton like any other function, passing any context-specific sequential code and any additional parameters as arguments. For example, the task farm skeleton in Figure 1.1 applies a given skeleton or sequential function to each input in parallel. The farm also takes the

number of workers as a parameter, allowing the programmer to control the amount of parallelism that is introduced. Since skeletons can be composed and nested, a wide range of parallel structures can be represented by a small set of skeletons. Moreover, a skeleton can have multiple implementations, including those for GPUs [70] and or high performance systems [113, 42] This makes swapping between the different implementations as simple as calling a different function, and equivalences between configurations of skeletons can be used to further improve performance [31]. In functional languages, skeletons are often implemented as *higher-order functions*. For example, the task farm skeleton in Figure 1.1 has implementations in the Strategies library (`parList`), the Par Monad (`parMap`), and the Accelerate library (`map`), amongst others.

Recursion schemes describe how data structures can be traversed or constructed [90]. The well-known `map`, for instance, applies a function to each element in a data structure without changing the spine of the structure itself. In functional languages, recursion schemes are often implemented as higher-order functions. For example, the standard `map` over *cons*-lists can be defined in Haskell as

```
1 map : (a -> b) -> [a] -> [b]
2 map f [] = []
3 map f (x:xs) = f x : map f xs
```

Recursion schemes are generic descriptions and so can be implemented for arbitrary types; e.g. binary trees, rose trees, and octrees. Since many parallel patterns have sequential recursion scheme equivalents, e.g. the task farm and `map`, it follows that we can use instances of sequential recursion schemes as *loci* of potential parallelism. A scheme can then be swap-replaced for an equivalent parallel pattern, just as the introduced pattern can then be swap-replaced for other equivalent parallel patterns or configurations thereof. For example, given the simple Sudoku solver [86]

```
1 sudoku ps = map solve ps
```

where `ps` is a list of puzzles, and `solve` is a function that solves a given puzzle, we can swap the call to `map` for a call to, e.g., the Par Monad:

```
1 sudoku ps = runPar $ parMap solve ps
```

Here, `parMap` is the task farm skeleton that applies `solve` to each element in `ps` in parallel, and `runPar` lifts the result out of the `Par` Monad. By taking advantage of equivalent recursion schemes and parallel patterns, parallelism can, in principle, be introduced quickly, easily, and (potentially) automatically and in a way that can take advantage of a range of architectures and parallel structures.

Despite the advantages of abstraction that are provided by both pattern-based parallelism and functional programming, introducing parallelism can still be a non-trivial task. Whilst we can in principle replace recursion schemes with their parallel equivalents, parallelising all recursion scheme instances in a program does not guarantee good parallel speedups. Choosing the recursion schemes, or *loci* of potential parallelism that are best suited for parallelism, and which of the potentially infinite equivalent configurations of patterns will produce good speedups, can be guided by the use of cost models or by profiling tools [32]. Another, more fundamental, problem is that there is no guarantee that the programmer will use recursion schemes in all possible instances, or that the best recursion scheme will be used. This means that opportunities for parallelism may be limited, including the discovery of equivalent parallel structures that may confer performance gains.

1.2 Automatic Pattern Discovery

In order for recursion schemes to be effective and optimal *loci* of parallelism, they must *i)* be used in the source code, *ii)* be used in the *right places*, and *iii)* be the recursion scheme that best represents the traversal behaviour. Despite the prevalence of higher-order functions in functional programs, there is no guarantee that all programmers will always choose to use recursion schemes over explicit recursion. For example, the *Spectral* set of the *NoFib* suite of Haskell benchmarks comprises a total of 48 programs [99]. Manual inspection shows that at least 19 (39.6%) of these have one or more functions that could be rewritten in terms of `map` or `fold` patterns alone. There are a number of reasons why a

programmer may not use a valid recursion scheme. For example, the programmer may not know a certain recursion scheme exists. Whilst `map`, `foldr`, and `foldl` are well-known recursion schemes, other schemes are less well-known; e.g. `scanl`, `unfold`, or one of the many general *morphisms* [59]. Alternatively, the choice may be deliberate: an explicitly recursive function may express its behaviour more clearly. For example, `wc`, as defined in [59], calculates the number of words in a string,

```
1 wc :: [Char] -> Int
2 wc cs =
3   para (\x xs z ->
4         if not (isSpace x)
5           && (null xs || isSpace (head xs))
6         then z+1
7         else z) 0 cs
```

Here, `para` is a *paramorphism*, a less common recursion scheme where the fixpoint function, i.e. the anonymous function passed to `para`, has access to both the results, `z`, of the recursive calls but also to the substructures, `xs`, that produced `z`. To understand `wc`, the programmer must be familiar with the definition of `para` before understanding the fixpoint function. We can define `wc` to use explicit recursion,

```
1 wc :: [Char] -> Int
2 wc [] = 0
3 wc (c:cs)
4   | not (isSpace c) && (null cs || isSpace (head cs))
5   = wc cs + 1
6   | otherwise
7   = wc cs
```

Here, it is immediately apparent that `wc` is using *structural recursion* [68], and so the programmer only needs to understand the guard expression as before. Similarly, the starting point of any program is to determine *what* the program needs to do and *how* this can be expressed. Explicitly recursive functions can be a simple means to check whether an idea works, and they can be easily adjusted if they do not. Conversely, recursion

schemes may require the programmer to significantly reformulate the problem. Once the program is functionally correct, the programmer may then transform their code to introduce relevant recursion schemes.

A manual approach to pattern discovery can be non-trivial, error-prone, and subject to the same reasons why recursion schemes are not used in the first place. The programmer may not realise that a function can be expressed as a particular scheme; they may know that a relevant scheme exists, e.g. `unfold` or `para`; and they may not introduce the scheme that accurately describes the behaviour of a given function. It follows, therefore, that if pattern discovery can be automated, and if all possible recursion schemes can be found and introduced, then more opportunities for parallelism are available while reducing the chances for introducing errors. Classical approaches to automatic pattern discovery have focussed on program calculational approaches. These are based on the Bird-Meertens Formalism [15] and equational reasoning [101] techniques, and reformulate function definitions into a desired pattern.

Most program calculation approaches use the Third List Homomorphism theorem [46], which states that if an operation can be expressed as both a leftwards-fold and a rightwards-fold, then that operation can be performed as part of a divide-and-conquer (parallel) pattern. For example, a function to sum a list of numbers, `sum`, can be defined as both a leftwards- and rightwards-fold:

```
1 sum = foldl (+) 0
2
3 sum = foldr (+) 0
```

This indicates that `(+)` is *associative*, and so may be performed as part of a divide-and-conquer pattern; e.g.

```
1 sum [x] = x
2 sum (xs ++ ys) = (sum xs) + (sum ys)
```

Here, `xs` and `ys` are two halves of the input list and are both passed to recursive calls, the results of those calls are then summed together using `(+)`.

Earlier list homomorphism approaches are limited in that they can only inspect functions that traverse lists, that the programmer must supply two versions of the same function (i.e. a leftwards- and a rightwards-fold), and that operations that do not prove to be associative cannot be used in a divide-and-conquer pattern. Later approaches seek to address these limitations. For example, Morihata *et al.* demonstrate how to generate a rightwards-fold from a leftwards-fold [94], and how the approach can be extended to trees, using upwards- and downwards-folds [93]. Elsewhere, Geser and Gorlatch rewrite *almost-homomorphisms* as homomorphisms using *anti-unification* [45]. Despite the improvements to the approach these make, they are themselves limited. Deriving rightwards-folds requires the leftwards-folds to be in a stylised form that limits the functions to which the technique can be applied. The generalisation to trees converts those trees to *zipper*s [67], i.e. lists of subtrees, which introduces potentially significant overheads. Finally, Geser and Gorlatch's approach shares the above limitations since it requires that both leftwards and rightwards definitions are provided in the form of *cons*- and *snoc*-lists, and is defined for list operations only. Another limitation of the list homomorphism approach is that it only finds one pattern; i.e. divide-and-conquer. Specialisations of the pattern, such as `map` or `scanl`, must be derived as an additional step to take advantage of their parallelisations. Similarly, patterns with different structures cannot be found. For example, `unfold` is a corecursive pattern that allows the guarded generation of arbitrary data-structures [47].

Other calculational approaches to automatic pattern discovery address these limitations by using a more general pattern: *hylomorphisms*. These are also divide-and-conquer patterns, but are not limited to lists, or a particular type. Hylomorphisms are comprised of `fold` and `unfold` parts, and can be used to express a wide range of patterns. Moreover, the laws that govern hylomorphisms can be used to calculate equivalent parallel pattern configurations [31]. Unfortunately, hylomorphisms in this form require highly stylistic code that is unlikely to be written by hand [61]. Hylomorphisms must therefore be discovered in arbitrary code, specifically requiring the discovery of `fold` and `unfold` patterns. Specialisations of `fold` and `unfold` can also be found and represented as

hylomorphisms. Little work has focussed on discovering hylomorphisms in arbitrary code, with approaches often limited by the functions that they can inspect and rewrite, and the inability to discover useful specialisations of `fold` and `unfold`.

There are few non-calculational approaches to automatic pattern discovery. One approach, introduced by Ahn and Han, uses *program slicing* to categorise subexpressions in recursive functions, where those categories denote whether the calculation can be performed as part of a particular pattern [2]. While this approach can discover a wider range of patterns than other approaches, it is nevertheless limited by the functions that it can analyse since they must be defined in a certain way.

Another non-calculational approach, introduced by Cook [41], uses *higher-order unification*. This approach compares the structure of two functions to infer whether the inspected function is an instance of a pattern that is defined by the second function, and if so, the arguments needed to express the inspected function as that pattern. This approach requires functions to be defined in a very stylistic way, thus reducing the range of functions to which it can be applied. Since the higher-order unification is performed by a proof assistant, any inspected code must be translated into the language of the proof assistant, then back again once analysis is complete. The approach is further limited in the range of functions to which it can be applied and the patterns it can discover, since it cannot e.g. inspect functions that use simultaneous induction (e.g. `zipWith`) or corecursive functions (e.g. `unfold`). Finally, higher-order unification itself is *undecidable*. Semi-decidable variants exist, but these further limit the range of functions to which the approach can be applied.

In this thesis, we build upon existing automatic approaches to pattern discovery. In Chapter 4, we introduce a novel program slicing approach that can inspect arbitrary pure functions. This expands the range of functions that can be analysed by current approaches. In Chapter 5, we introduce a novel anti-unification approach that can discover arbitrary *recursive* patterns and a limited set of *corecursive* patterns in pure functions. Additionally, this approach is *user-extensible* such that patterns to be discovered can be provided by the programmer. This expands the the range of patterns that can be automatically discovered by existing

techniques. Together, our approaches are able to discover more patterns in a wider range of functions than previously possible. Consequently, our approaches are able to discover more *loci* of potential parallelism, and so simplify and further automate the parallelisation process.

1.3 Contributions

This thesis makes the following novel contributions.

1. In Chapter 4, we introduce a novel approach that can discover mappable computations in arbitrary pure recursive functions using *program slicing* techniques.
2. In Chapter 5, we introduce a novel approach to determine whether a given function can be rewritten as a call to a given recursion scheme implementation, and if so, to derive the arguments to that recursion scheme that are needed to maintain functional correctness. We use *anti-unification* techniques as the foundation of this approach. Our technique is able to discover instances of schemes that can be represented as `fold` patterns. We demonstrate how our approach can be extended to discover schemes with different pattern structures.
3. As part of our approach defined in Chapter 4, we define a novel *program slicing* algorithm, where inclusion in a slice indicates that a variable is either *used* or *updated* between successive recursive calls. We prove that our algorithm is sound with respect to our definition of a slice and our assumptions, and that the calculated slice is unique for each recursive function and slicing criterion.
4. As part of our approach defined in Chapter 5, we define a novel *anti-unification* algorithm that compares the structures of two functions in order to infer candidates of pattern arguments. We prove that our algorithm is sound with respect to the syntactic equivalence property that we define based on the standard syntactic equality property of traditional anti-unification algorithms.

5. We have implemented prototypes for both *slicing* and *anti-unification* approaches to pattern discovery, which we describe in Sections 4.4 and 5.5 respectively. Our prototype for the *slicing* approach is implemented as a stand-alone Erlang program, and our prototype for the *anti-unification* approach is implemented as an extension to the Haskell refactoring tool, HaRe [18].
6. We have evaluated both of our pattern discovery approaches on a range of examples. These include five standard parallel benchmarks, including benchmarks from the NoFib suite [99]. Other examples include examples from the Haskell Prelude, and examples from Cook and Kannan’s theses [41, 74]. We demonstrate that our approaches are able to discover all instances of a range of common patterns, including `map`, `foldr`, `foldl`, `scan`, and `zip`. We present our results in Sections 4.6 and 5.7, respectively.
7. We give the average times taken by our prototype implementations for all examples. Our *program slicing* approach is able to discover all mappable computations in our largest example in under 0.7ms, and our *anti-unification* approach is able to discover all patterns in our largest example in under 2s. As a synthetic benchmark, we have tested our prototypes with varying sizes of input. Our results suggest that the prototype implementation of our *program slicing* approach runs in quadratic time with respect to the number of operations classified. Similarly, our results suggest that the prototype implementation of our *anti-unification* approach runs in exponential time with respect to the number of function clauses of both the inspected function and the given pattern, but also that our prototype implementation runs in linear time with respect to the number of functions and patterns that are compared.
8. In section 4.6.2, we present parallel speedup results for our parallel benchmarks. For all our experiments, we achieve maximum speedups of $32.93\times$ sequential runtimes on our 28-core (56-core with hyperthreading enabled) experimental machine, *corryvreckan*. This

shows that standard recursion schemes can be used to introduce parallelism and that they can produce good speedups.

1.4 Publications

Some of the work presented in this thesis has been previously published in a number of journal and workshop papers. In this section, we give an overview of these papers and describe how this thesis expands upon them.

In Search of a Map: Using Program Slicing to Discover Potential Parallelism in Recursive Functions, Workshop on Functional High-Performance Computing (FHPC), co-located with ACM International Conference on Functional Programming (ICFP), Oxford, UK, pp 30–41. A. D. Barwell and K. Hammond. In this paper, the author developed and presented the program slicing approach that is described in Chapter 4. Results are also presented. The author wrote the paper with K. Hammond assisting with editing, and advised upon the presentation and correctness of definitions. Chapter 4 extends this work by defining a refactoring to introduce `map` operations, and presents additional examples.

Finding Parallel Functional Pearls: Automatic Parallel Recursion Scheme Detection in Haskell Functions via Anti-Unification, Future Generation Computer Systems, 2017. A. D. Barwell, C. Brown, and K. Hammond. In this paper, the author developed and presented the anti-unification approach that is described in Chapter 5. Results, including those of the parallel benchmarks, are included here. The author wrote the paper with C. Brown assisting with editing the initial submission, and K. Hammond assisting with editing the final version. Chapter 5 extends this paper by: *i*) describing our prototype implementation; *ii*) introducing an approach to discover instances of the `unfold` pattern; and *iii*) presents additional examples, including an analysis of the time complexity of our prototype.

Using Program Shaping and Algorithmic Skeletons to Parallelise an Evolutionary Multi-Agent System in Erlang, Computing and Informatics 35(4), pp 738–818, 2016. A. D. Barwell, C. Brown, K. Hammond, W. Turek, and A. Byrski. This paper describes a case study that shows how refactoring techniques can introduce parallelism in an Evolutionary Multi-Agent System (EMAS). In this paper, we define novel refactorings, and apply those refactorings to the EMAS, producing a parallel version. The author defined and implemented refactorings to transform the code from sequential to parallel versions. The EMAS itself was developed and described, and parallel results were provided by co-authors at the AGH University of Science and Technology in Krakow, Poland. The author wrote the paper with the exception of the background section explaining the EMAS, and the majority of the results section, outlining parallel results. The experiments were also run by AGH. C. Brown and K. Hammond assisted with editing the paper. We cite this paper in Chapter 2 as an example of how refactoring techniques can facilitate the introduction and manipulation of parallelism.

Towards Semi-Automatic Data-Type Translation for Parallelism in Erlang, ACM Erlang Workshop 2016, Nara, Japan. A. D. Barwell, C. Brown, D. Castro, and K. Hammond. This short paper proposes a refactoring approach to the semi-automatic conversion of Erlang data-structures of one type to an equivalent data-structure in another type. The proposed approach uses program slicing to ensure that all expressions that depend upon the translated data-structure are themselves translated accordingly. The author developed the translation approach and wrote the original full paper, with editorial assistance from C. Brown and K. Hammond. Once accepted as a short paper, K. Hammond edited the original full paper to its current length and form, taking information only from the original paper. D. Castro presented the work at the workshop. We cite this paper in Chapter 2 as an example of how static analysis techniques can be used to guide the refactoring process, and ultimately tune parallel performance.

The Missing Link! A New Skeleton for Evolutionary Multi-Agent Systems in Erlang, IJPP, 2017. J. Stypka, W. Turek, A. Byrski, M. Kisiel-Dorohinicki, A. D. Barwell, C. Brown, K. Hammond, and V. Janjic. This paper describes the EMAS in [11] as a generic meta-heuristic framework that may be instantiated for different context-specific examples. The author contributed only the experimental results, and the descriptions thereof. AGH and C. Brown wrote and prepared the majority of the paper, with additional editing by K. Hammond.

Discovering Parallel Pattern Candidates in Erlang, ACM Erlang Workshop 2014, pp 13–23, Göteborg, Sweden. I. Bozó, V. Fordós, Z. Horvath, M. Tóth, D. Horpácsi, T. Kozsik, J. Közegi, A. D. Barwell, C. Brown, and K. Hammond. This paper reports on the ParaPhrase Refactoring Tool for Erlang. The tool is designed to inspect Erlang source code for parallelisable computations. Once found, the tool introduces algorithmic skeletons from the Skel library [70], and profiles the produced parallel code in order to find and introduce the optimal parallel configuration. The writing of this paper was led by our co-authors at Eötvös Loránd University in Hungary; the author contributed to the background section on algorithmic skeletons and the development of the Skel library. We discuss this technique in Chapter 3, and explain its relation to the work presented in this thesis.

Using Erlang Skeletons to Parallelise Realistic Medium-Scale Parallel Programs, High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU) 2017, Austria. V. Janjic, A. D. Barwell, and K. Hammond. This paper reports on parallel speedup results achieved using the algorithmic skeleton library, Skel, on three medium-scale Erlang programs. The writing of this paper was lead by our colleague Vladimir Janjic at the University of St Andrews; the author contributed to the introduction, conclusions, and background sections. The author also contributed parallel results for the Image Merge example, and to their presentation in the paper.

Existing Parallelisation Approaches

For decades, Moore’s Law has allowed programmers to benefit from performance improvements for free [43]. However, as sequential processors have tended towards the physical limits of miniaturisation, we have turned to replication in order to continue to find improvements to performance [52]. Consequently, parallel hardware is now ubiquitous [11]. Unfortunately, and unlike sequential processors, adding more cores does not confer free performance improvements to programs [43]. The ability to take advantage of parallel hardware, and thus the potential improvements to performance they represent *via* parallel programming, has become a necessary skill for expert and average programmers alike. Unfortunately, and unlike sequential programming, taking advantage of parallel hardware requires the programmer to consider additional coordination aspects; e.g. how a computation is decomposed, communicated among, and synchronised across available processing elements [14]. Moreover, since parallel hardware is increasingly *heterogeneous* [8, 42, 70], and that each architecture has its own requirements, such as GPUs needing to apply the same instructions to batches of data, writing parallel programs that take advantage of all available hardware can further complicate an already difficult parallelisation process [70].

Traditional approaches to parallelism, such as pthreads [28] or Parallel STL in C++ [88], consist of low-level parallel primitives added to existing

programming languages. They require the programmer to consider all coordination aspects of parallelism. Due to the complexity of this approach, these approaches are highly susceptible to errors; such as race conditions, deadlocks, and livelocks [11, 14, 52]. In response, a range of techniques have been developed to facilitate parallelisation, including both fully-automatic and semi-automatic approaches, which we describe in this chapter.

2.1 Automatic Approaches to Parallelisation

Automatic approaches to parallelisation seek to simplify parallelisation for the programmer by removing the programmer from the equation. The majority of automatic approaches focus on introducing low-level parallel primitives transparently at compile-time in imperative languages. Amongst others, these include: Lamport’s parallelisation of Fortran `do`-loops [77]; Burke and Cytron’s improvements to the detection of nested loops [26]; and Artigas’ approach for Java’s loops [5]. These approaches generally use data-dependency information to work out which loops can be parallelised, combined with profiling information to determine whether parallelisation would be profitable [109, 116].

More recent approaches for automatic parallelisation have focussed on the use of GPUs [54, 80] and the *polyhedral model* [6]. The polyhedral model is a loop optimisation and parallelisation approach that represents each stage in a (potentially nested) loop as a point in a lattice. This exposes dependencies between lattice points, and so enables the reordering of statements in the analysed loops to safely optimise their operation and introduce parallelism [13]. One limitation of the polyhedral model, is that it requires the AST of the loop to be translated into a specific linear-algebraic form, and not all statements can be translated to this form [13, 53].

Despite being generally desirable, automatic approaches to parallelisation are limited in a number of ways. Since these approaches are dependent on complex program analyses, it follows that they can only inspect the patterns in code to which their analysis can be applied. Moreover, extend-

ing these analyses can be difficult; any program transformations must be correct for all cases, for example. An automatic parallelising tool must be *trustworthy* since the programmer cannot influence the introduced parallelism; they should therefore introduce no errors, and they should improve the performance of the parallel program. Another limitation of automatic approaches is that they can often only introduce one type of parallel structure, and normally for one type of architecture. Approaches that use GPUs can expand the range of architectures automatic parallelisation approaches can use, but are still limited by both the structures they can analyse and the structure of parallelism they can introduce. Moreover, since automatic approaches often introduce low-level parallel primitives, it can be difficult to easily swap between equivalent parallel implementations. These limitations can reduce the amount of parallelism that can be introduced to a program, and so potentially reduce any performance gains that can be achieved.

2.2 Semi-Automatic Approaches to Parallelisation

An alternative is to use *programmer-in-the-loop*, or semi-automatic, approaches. These can enable the programmer to see what parallelism is being introduced, but also to affect the outcome. For example, in 1991, Kennedy *et al.* developed an editor that allowed the programmer to inspect control and data dependencies in Fortran code and then introduce parallelism via *refactorings* [75]. Since these approaches can use the programmer as an oracle, Udupa *et al.* take a more speculative approach to parallelisation, where dependencies are broken without guarantees of correctness [114]. Instead, the programmer must inspect the suggested changes, and determine whether functional correctness is maintained. Since these approaches introduce parallelism via low-level parallel primitives and libraries, it remains difficult to swap between parallel implementations and architectures. An alternative approach is to raise the level of abstraction that is presented to the programmer.

Functional programming has a long history with parallelism [12] and can be used to abstract how programs are evaluated, so providing a clear separation between algorithmic code and its evaluation behaviour [86]. *Higher-order functions*, in particular, facilitate the abstraction of behaviour since e.g. context-specific functions can be accepted as input to functions, such `map`, that denote how a data-structure is traversed, or how an operation should be evaluated. Haskell, for example, is a lazy functional language, meaning that evaluation to normal form occurs only when a value is needed. The language is extended using Futures [86], represented by the `par` and `seq` operators, that allow the programmer to indicate that a given operation can be performed in parallel. This concept is not limited to functional languages, a similar technique can be found in OpenMP's *pragmas*, for example, where regions of code are declared as parallelisable [34]. This allows the parallelism to be introduced transparently to the programmer, whilst still allowing the programmer to guide how parallelisation occurs. Similarly, the Task Parallel Library [79] automatically maps tasks to threads, but the programmer is responsible for splitting and synchronising the data. Another approach by Calderon *et al.* uses strictness analysis to automatically activate and deactivate, and in some cases insert, `par` and `seq` operations in Haskell source code [110]. Both `par` and `seq` operators can be further abstracted in order to increase the level of abstraction of parallelism presented to the programmer. The Strategies Library [111], the Par Monad [87], NESL [16], GUM [112], Eden [85], and the Accelerate library for GPUs [33], all present higher-level primitives for introducing and manipulating parallelism. These approaches allow the programmer to better control *when* and to what level (e.g. normal form) an operation will be evaluated. While these approaches are higher-level than traditional parallel primitives and libraries, they still require the programmer to coordinate the mapping of tasks to threads and communication between those threads.

These coordination aspects can be abstracted away by using (embedded) domain-specific languages (DSLs). Feldspar [8] is one approach for Haskell that is designed to allow the programmer to describe high-level and platform-independent descriptions of digital signal processing (DSP) algorithms, compiling them to C. Similarly, RPL by Janjic *et al.*

is designed to express the (parallel) structure of a C++ program in a simplified way, effectively separating what the program does from how it does it [71]. RPL is extended with a series of refactorings that work either directly on the C++ source code, or on its RPL representation. To guide the application of these refactorings, *recipes* are presented to the programmer to suggest how an optimal parallel configuration can be introduced. Aldinucci's META tool [3] is similar to RPL, but instead presents the (parallel) structure of a program as a graph. Transformations applied to the graphical representation are similarly applied to the underlying source code. The META tool is presented for a generic target-language. DSLs present a simplified interface that the programmer can use to reason about structure and program transformations when introducing parallelism. Whilst this is advantageous, it follows that the source and target language likely need to be transformed to the DSL, which must be expressive enough to describe all possible parallelism in the source and target language(s), and whose syntax must be learnt by the programmer. If the high-level information exposed by DSLs can be encoded into the language itself, these potential problems can be avoided.

One technique that achieves this is the concept of *algorithmic skeletons*. Introduced by Cole in 1988 [38], skeletons are a structured approach to parallelism that present implementations of high-level patterns to the programmer. Skeletons abstract away low-level mechanics of parallelism that are a common source of error; e.g. process management, communication, locks, and synchronisation [39]. Since skeletons can be nested and composed, even a small set of skeletons can represent a wide range of parallel structures [31]. Moreover, a single skeleton may have multiple implementations, including implementations in various languages [12], and for *heterogeneous* [70] or *high-performance* parallel systems [42], which may involve massive numbers of processors, of possibly different types.

In functional languages, skeletons are often implemented as *higher-order functions* [9]. This allows them to be initialised with any context-specific sequential code, nested skeleton instances, and parameters that are required by the skeleton. Implementations of skeletons may be collected in *algorithmic skeleton libraries*, allowing the programmer to call skeletons like any other function. A wide range of skeleton libraries

have been developed for a number of programming languages since their introduction in 1988. Some skeleton libraries include: Skel for Erlang [69]; the Strategies library [111], the Par Monad [87], Eden [85], and Feldspar [8] for Haskell; FastFlow [72] for C++; Microsoft's Parallel Patterns Library [29]; and Intel's Threading Building Blocks library [40]. Other skeleton libraries are available, with the study by González-Vélez in 2010 listing at least 22 [50]. Similarly, a range of skeletons has been developed; including, but not limited to: a *pipeline* skeleton, a *task farm*, and a *feedback* skeleton.

A parallel *pipeline* (Figure 2.1) applies a sequence of skeletons, s_1, \dots, s_n , in turn to a stream of independent inputs, x_1, \dots, x_n , where the result of s_i is passed as input to s_{i+1} . Parallelism arises from the fact that each $s_i(x_j)$ can be performed in parallel with $s_{i+1}(x_{j+1})$. A *task farm*, or *parallel map*, (Figure 2.1) applies a skeleton, s , to each element in a stream of independent inputs, x_1, \dots, x_n . Parallelism arises from the fact that each $s(x_i)$ can be performed in parallel with all other $s(x_j)$. Finally, *feedback* allows looping operations. It repeatedly applies a skeleton, s , to each independent input in a stream, until a given condition is reached. Skeletons can additionally represent context-specific sequential code, usually in the form of functions. Consequently, nested skeletons are either: *i*) context-specific sequential code, or *ii*) another skeleton instance. Other skeletons are possible, including skeletons that can be formed as combinations of other skeletons; e.g. Divide-and-Conquer [50], Branch and Bound [50], Bulk Synchronous Parallelism (BSP) [97], and the butterfly pattern [96].

Despite the benefits of abstraction, skeletons are limited by the requirement that the programmer knows *which* skeletons should be called, *where* they should be called, and *what* parameters should be passed to them. The RPL approach by Janjic *et al.* [70], uses profiling information to determine the optimal configurations of skeletons for a given program, and then suggests how they might be introduced via the built-in refactorings [70]. However, this approach requires the use of the DSL itself, and the recipes only focus on the immediate introduction and configuration of the skeletons. Other approaches also use profile information to suggest where in the source code a program might benefit from parallelism,

2.2. Semi-Automatic Approaches to Parallelisation

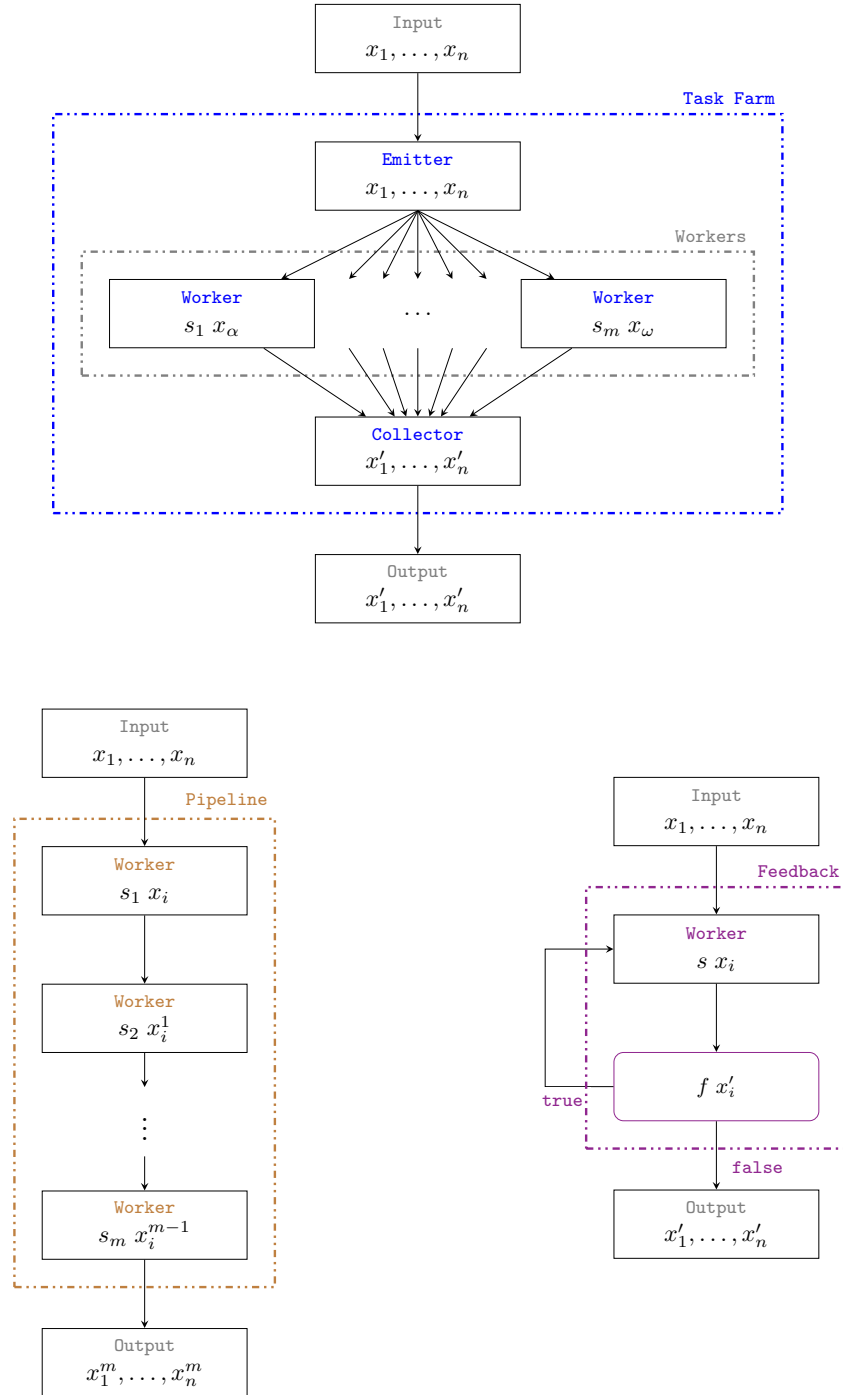


Figure 2.1: Three example algorithmic skeletons: a parallel *pipeline*, a *task farm*, and a *feedback*.

and with what configuration of skeletons. Hammond *et al.* present a profiling approach to automatically derive skeleton configurations for EDEN, using Template Haskell to automatically change configurations at runtime [57]. Similarly, Scaife *et al.* developed a parallelising compiler for SML that replaced `map` and `fold` operations by equivalent skeletons, using profiling to determine which `map` and `fold` operations should be replaced [107]. More recently, Castro *et al.* have combined dependent types, the laws of hylomorphisms, and cost models to automatically derive the optimal configuration of skeletons [31, 32].

While profiling and cost models can be used to suggest *where* in the source code a program might benefit from parallelisation, and can be used to determine an optimal configuration of skeletons, these approaches do not consider how an arbitrary program might be restructured to facilitate, or even *enable*, the introduction of skeletons. Recent work considers how *refactoring* techniques can be used to facilitate the restructuring of source code to introduce skeletons.

The term refactoring was introduced by Opdyke in his thesis [98], but the concept goes back as far as Burstall and Darlington’s fold/unfold system [27]. Refactoring commonly refers to the act of changing the internal structure of a program without changing its functional behaviour [19]. While such transformations can be performed mechanically, a range of refactoring tools have been developed for a number of languages and IDEs [17, 18, 58, 60, 71, 83, 91]. This reduces the opportunity for error since a refactoring tool can ensure that the correct code is transformed. Whilst the majority of refactoring tools have been developed for imperative languages, some refactoring tools do exist for functional languages. These include: Wrangler [83] and RefactorErl [17] for Erlang, and HaRe [18] and Haskell Tools [58] for Haskell. Common refactorings include: *rename*, where a variable, function, or module is renamed, and all instances are correctly changed; *lift function*, where a new function is declared and the selected code is lifted into the body that function; and *inline*, which takes a function or variable name and replaces an instance of that name with its definition.

Refactoring can be seen as an abstraction over low-level transformations, and individual refactorings can be combined to produce more

complex refactorings [81]. In [43], Dig discusses how refactoring can be used to facilitate the introduction and management of parallelism. A similar approach has been taken by Brown *et al.* in [19], where they introduce and manipulate skeletons using a series of refactorings and the Erlang refactoring tool, Wrangler. Brown *et al.* define four novel refactorings to introduce three skeletons using the Skel library, and to group inputs to increase granularity. Cost models are used to evaluate how well the introduced parallelism performs. Refactorings can be applied multiple times, and can be undone should a refactoring reduce parallel speedups. Both refactorings and Skel were extended in 2015 by Janjic *et al.* to enable the use of GPUs [70]. Here, Skel is extended with skeletons that utilise GPUs, and refactorings are defined to introduce the new skeletons, and also to generate the GPU kernel code. This approach is not limited to Erlang, with Brown *et al.* demonstrating how refactoring can be used to introduce skeletons in Haskell [22] and C [20].

These refactorings focus primarily on the immediate introduction and manipulation of skeletons. Since parallelisable code may not be immediately obvious, it is often necessary to first restructure code to facilitate, or even enable, the introduction of skeletons. In [11], the authors demonstrate how standard refactorings can be combined with those defined by Brown *et al.* in order to restructure an Evolutionary Multi-Agent System in Erlang. While this demonstrates how refactorings can be combined to introduce parallelism, it does not discuss how a programmer should know *which* refactorings to apply, *where* to apply them, and in *what* order. External guides, such as documentation or design patterns in object-oriented languages [103], are thus required. Li and Thompson demonstrate how to introduce patterns of concurrency in Erlang programs using *program slicing* to guide the refactoring [82]. This approach does not use algorithmic skeletons but the concurrency primitives of Erlang, and does not provide any guarantee that any concurrency can lead to parallel speedups. Similarly, the authors in [10] suggest how program slicing can be combined with refactoring to direct type transformations in Erlang programs. While this approach can *tune* parallelism, it does not introduce it or discover where it might be introduced.

2.3 Summary

Parallelisation allows programmers to take advantage of the increasingly parallel and heterogeneous systems that are now ubiquitous in order to increase the performance of programs. Simplifying traditional, low-level approaches to parallelisation has been the focus of much work, including both fully-automatic approaches described in Section 2.1 and semi-automatic approaches described in Section 2.2.

For this thesis, we have chosen to use algorithmic skeletons for parallelisation since skeletons support a wide range of parallel structures and can be expressed using a small set of patterns. Additionally, skeletons can have multiple implementations, and so both a wide range of hardware and a wide range of underlying approaches can be taken advantage of easily. This also facilitates switching between different implementations and architectures. Since parallel patterns are language independent, any techniques that build upon skeletons are also more likely to be language independent, as well as simplifying the understanding of parallel program structures. Relatedly, and since skeletons can be reasoned about, the optimal configuration of skeletons can be discovered automatically. Finally, since skeletons are an *explicit* approach to parallelism, the programmer can inspect and affect the introduction of parallelism, meaning the approach is more flexible than fully-automatic techniques.

Despite the advantages of algorithmic skeletons, they must nevertheless be introduced to the program, a problem that is often non-trivial and is not considered by skeleton approaches. Although refactoring techniques can be used to aid the restructuring of programs in order to facilitate the introduction of skeletons, as the number and complexity of refactorings increases, the original problem of introducing refactorings is merely converted to a problem of knowing which refactorings to use. Consequently, alternative approaches must be sought to automatically determine where in a program that skeletons can be introduced. The automatic identification of *loci* of potential parallelism will simplify and further automate the parallelisation of a wider range of programs, allowing potentially significant performance gains to be achieved simply and automatically.

Automatic Pattern Discovery for Parallelisation

In order to facilitate and further automate the introduction of parallelism *via* algorithmic skeletons, we must overcome the challenge that is determining *where* in a program that they can be introduced. Since skeletons are often implemented as higher-order functions in functional languages, we can take advantage of equivalences between skeletons and sequential recursion schemes, such as `map` or `fold`. Instances of recursion schemes can be treated as *loci* of potential parallelism, and can be swap-replaced with calls to equivalent skeletons to introduce parallelism. The decision of whether a scheme instance, or pattern, is worth parallelising can be determined through the use of cost models or profiling techniques [107]. Moreover, we can also determine an optimal parallel configuration using equivalences between skeletons, or parallel patterns, to enumerate a range of possible options, that can be compared using cost models or profiling techniques [31, 32]. While this is possible using an arbitrary pattern, since some patterns are specialisations of others, such as `map` and `foldr`, and since specialisations more closely reflect the traversal behaviour of a specific instance than more general patterns, it follows that it is desirable to choose specialised patterns over more general patterns. This encoded information can then be used by e.g. [31] to more accurately discover an optimal parallel configuration. Since there is no guarantee that all possible patterns will occur in a given program, or that specialisations

will always be chosen over their more general forms, it is useful to detect patterns in code *automatically*. A number of approaches to automatic pattern discovery currently exist. The majority of these have their roots in the Bird-Meertens Formalism [15], and calculate instances of patterns from functions. We refer to these as *program calculational* approaches, and discuss these in Section 3.1. Non-calculational approaches also exist, and we discuss these in Section 3.2.

3.1 Program Calculation Approaches

Since functional languages provide a good foundation for equational reasoning [115], a number of approaches have been developed based on the Bird-Meertens Formalism [15] to transform programs using a *calculational* approach. This is similar to the Worker/Wrapper transformation technique described by Gill and Hutton [49], which allows the type of data structures and operations to be transformed to allow more efficient accesses. Most calculational approaches to automatic pattern discovery are based on the Third List Homomorphism theorem, or the discovery of another general recursion scheme, *hylomorphisms*. We discuss both in the below sections.

3.1.1 Using the Third List Homomorphism Theorem

The Third List Homomorphism theorem states that given functionally equivalent leftwards and rightwards folds that express the same operation, then the operation is associative and commutative, and can therefore be performed as part of a divide-and-conquer pattern [46]. In this basic form, *list homomorphisms* require two sequential `fold` implementations of the same operation. This is limiting in two ways: *i)* two `fold` definitions are required for the same operation; and *ii)* not all interesting list functions, such as *maximum segment sum*, are expressible as list homomorphisms. In order to address these limitations, Geser and Gorlatch present an approach that uses *anti-unification* to derive list homomorphisms from two pure explicitly recursive functions: one defined using *cons*-lists, and one defined using *snoc*-lists [45]. This approach is also

able to produce list homomorphisms from *almost homomorphisms* such as *maximum segment sum*. Since all functions that traverse a single list can be represented as almost homomorphisms [51], the ability to discover almost homomorphisms means that, in principle, all such recursive functions can be represented as a list homomorphism. Anti-unification compares the structure of two functions, to produce a third function, known as the *anti-unifier*, that represents the shared structure, and two sets of substitutions that represent the rewrites needed to restore the original functions from the anti-unifier. Geser and Gorlatch apply anti-unification twice, and use a calculational approach to derive function definitions that comprise the list homomorphism. While this approach is able to discover list and almost homomorphisms from explicitly recursive functions, it still requires two equivalent sequential functions, only applies to functions that traverse single lists, and the conversion of almost homomorphism to list homomorphism uses a projection function that introduces additional overheads.

Morita *et al.* address the requirement of two equivalent sequential definitions by calculating *weak right-inverses*, or rightwards folds, from leftwards folds [94]. While this simplifies some cases, leftwards definitions must be in a stylised form, and some of the requirements preclude the discovery of obvious examples; e.g. length since the approach assumes that result of the generated weak right inverse is a list of constant length. An alternative approach is presented by Hu, Iwasaki, and Takechi [62]. The authors use program calculation to discover list homomorphisms from a single explicitly recursive function that traverses a single list. The approach is able to discover almost homomorphisms from mutually recursive functions. The authors also claim that their approach is extensible to trees but do not demonstrate this. A similar approach is presented by Chin, Takano, and Hu, that also synthesises the projection function using calculational techniques [36]. Chin *et al.* also present a calculational approach to derive parallel forms for specialisations of *catamorphisms*, i.e. folds, and *paramorphisms*, i.e. folds that remember the substructures used to produce the accumulated result. Hu, Takeichi, and Chin extend this parallel form derivation to derive parallel functions from a wider range of function definitions, and are able to introduce both list homo-

morphisms and *mutumorphisms* [63]. A mutumorphism is a collection of functions that are mutually recursive. Whilst these approaches represent an improvement, they remain limited by the range of patterns that they can discover, and both derive and introduce parallel forms as part of their operation. The derived patterns are unique, and switching between different parallel implementations of patterns is likely to prove difficult. Once again, the authors of these approaches claim that the technique is extensible to data types other than lists, but this is not shown, and is likely to require significant time and effort of anyone who chooses to extend the approach.

Where the above techniques are based on calculational approaches that have been proved sound, Chi *et al.* synthesise list homomorphisms from proofs of associativity using a proof assistant [35]. Unfortunately, this approach means that it is difficult to extend to other languages since it would require translations between the program language and the proof assistant language.

One key limitation of approaches that are based on list homomorphisms is the range of patterns that they are able to introduce. Approaches thus far have only been able to introduce list homomorphisms, and in mutumorphisms in [63]. Mu and Morihata address this limitation, and extend the calculational approach to discover `unfold` instances for functions that generate lists [95]. This facilitates the translation of arbitrary data types to lists, and enables examples that handle sub-lists such as those found in the standard quicksort definition; i.e.

```
1 qsort []      = []
2 qsort (x:xs) = (qsort leq) ++ [x] ++ (qsort gt)
3   where leq = filter (=<x) xs
4           gt  = filter (>x) xs
```

Since quicksort is not structurally recursive, it cannot be described as a fold. Supporting `unfold` operations also enables data to be distributed more efficiently since data that can be generated can be sent to a process and then expanded to their full lists. This is demonstrated by Liu, Hu, and Matsuzaki by applying the the list homomorphism approach to Google's MapReduce skeleton, showing that the structure/divide-and-

conquer pattern introduced/enforced by homomorphisms can be useful and automatically adjusted for a range of hardware/systems [84].

Where the `unfold` pattern can be used to translate arbitrary types to lists, an alternative approach is to extend list homomorphisms to other types, such as trees. Morihata *et al.* generalise homomorphisms on trees, using upwards- and downwards-folds instead of the usual leftwards- and rightwards-folds [93]. This approach uses zippers [67] to divide trees into lists of subtrees and find the list homomorphism over that zipper. It could therefore be argued that the approach presented in [93] translates data types to lists, but as part of the homomorphism approach, instead of being left as an exercise to the programmer. Both `unfold` and zipper translation approaches introduce additional runtime overheads.

Conversely, Diffusion [64] has been shown to be extensible to binary trees without requiring translation. Simple diffusion is based on the third list homomorphism principle, but is described as being too limiting. The full diffusion approach can discover `map`, `fold`, and `scan` patterns using calculation. The approach can also be extended by combining it with external normalisation procedures, since explicit function definitions need to be presented in a stylised form in order to successfully find the recursion schemes. Although Diffusion expands the range of discoverable patterns to include specialisations of `fold`, it cannot discover more general divide-and-conquer patterns like other calculational approaches. Since Diffusion is also a deforestation technique, it may *remove* patterns from code in addition to exposing them. While this may optimise sequential programs, it can *reduce* opportunities for parallelism in parallel programs [36, 41].

A similar approach, introduced by Launchbury and Sheard, uses standard fusion techniques in combination with a dynamic set of rewrite rules to discover folds in source code [78]. Once again, extending this approach to other patterns is likely to be difficult, and whilst folds are useful, more specific forms, or other forms of patterns can also be useful in deriving the optimal configuration. Finally, and since fusion is another deforestation technique, it too can reduce opportunities for parallelisation.

The large amount of work on list homomorphisms has produced

a comprehensive approach to automatically discovering homomorphisms. Despite this, a number of general limitations remain. As the name suggests, list homomorphism approaches are primarily concerned with the discovery of list homomorphisms. Some approaches have extended this to other patterns, including mutomorphisms, specialisations such as `map`, and `unfold`, but these still represent a limited subset of patterns. Moreover, as these extensions demonstrate, extending the approach further to other patterns is a difficult challenge. Another limitation of list homomorphism approaches is the range of functions they are able to analyse. Whilst the ability to convert almost homomorphisms to list homomorphisms expands the range of analysable functions, this range is still limited to explicitly recursive functions that traverse a single list. Moreover, transforming almost homomorphisms to list homomorphisms introduces additional overheads. The extensions to trees and other patterns are similarly limited with respect to analysable functions, and also introduce additional overheads. Extensibility, range of analysable functions, and range of discoverable patterns all limit the number and range of patterns that can be discovered, and therefore limits possibilities for parallelism.

3.1.2 Hylomorphisms

A *hylomorphism* is a more general expression of a divide-and-conquer skeleton. They comprise a combination of an `unfold` and a `fold`.

$$\text{hylo}_F f g = f \circ F (\text{hylo}_F f g) \circ g$$

Here, g describes how the input is split (i.e. an `unfold`) and f describes how it is combined (i.e. a `fold`). The hylomorphism is indexed by the bifunctor, F , that defines the (un)folding operations. Morihata demonstrates how hylomorphisms can be used as a generalisation of the list-homomorphism theorem, and is used to produce good parallel speedups [92]. Similarly, Castro *et al.* demonstrate how the hylomorphism laws can be used to calculate equivalent parallel configurations [31], and when extended by cost models, can be used to discover optimal parallelisations [32].

Despite hylomorphisms being more general than list homomorphisms, and so allowing them to represent arbitrary data-types and a wider range of operations without producing additional overheads through projection functions and translations, they are not a standard recursion scheme. This limits their usefulness as *loci* of potential parallelism. Moreover, they are often represented in very stylised forms and/or programming styles, such as point-free. To address this, Hu, Iwasaki, and Takeichi present an approach to derive structural hylomorphisms from recursive definitions [61]. Entire functions are rewritten using program calculation techniques to represent the transformed functions as hylomorphisms. No inverse transformation is presented, and since the entire function is used, *near patterns* cannot be found. As a result, hylomorphisms cannot be transformed back into explicit recursive style, and some normalisation may be necessary in order to first discover those hylomorphisms. Moreover, this approach finds only hylomorphisms, and not any specialised forms, e.g. `map`. In Castro's thesis [30], he presents an alternative approach that automatically calculates the `fold` and `unfold` parts of hylomorphisms from an explicitly recursive function using the Applicative functor. Castro does provide a means to restore to a standard form, but again, this can only expose hylomorphisms that are in a specific form. Near patterns, or code that could be written as a hylomorphism, but isn't, will not be found.

Whilst hylomorphisms present a generalisation over list homomorphisms, the relative lack of work on hylomorphisms means that current approaches to discovering hylomorphisms are very limited. The range of functions that can be analysed and rewritten must be instances of hylomorphisms, and once again, only hylomorphisms can be discovered.

3.2 Non-Calculational Approaches

Three non-calculational approaches exist for automatic pattern discovery in functional languages. The first of these uses a combination of heuristics, profiling, and refactoring techniques to discover and introduce parallelism in Erlang code [17]. While, unusually, this approach works

for impure code, it is limited to discovering and introducing `map` and pipeline skeletons. The range of patterns it is able to discover is limited by the heuristics encoded into the system, where the presented heuristics are relatively simple near patterns. Moreover, this approach is limited to only two patterns. Since it introduces skeletons, it is at least simple to swap the introduced skeletons for alternative implementations. Extending this approach to other patterns is a difficult process, as demonstrated in [76], which extends the approach to discover divide-and-conquer patterns. Since this extension is based on the same heuristics approach, it shares the same limitations.

An alternative approach is presented by Ahn and Han in [2]. This approach uses *program slicing* to categorise subexpressions of explicitly recursive functions according to the pattern in which their computation may be performed. It is able to discover instances of `map`, `fold`, `scan`, and `zip` patterns for a first-order language. Extending the approach to other patterns is likely to be non-trivial. The functions this approach can analyse is similarly limited, where each constructor of the traversed data type must be matched by exactly one clause, and functions must have exactly one argument. One advantage of this approach, however, is its ability to analyse individual *expressions* and whether they can be performed as part of a pattern. All other current approaches to automatic pattern discovery analyse *functions*, such that entire functions are instances of a particular pattern.

The final non-calculational approach is similar to the synthesis-via-proof approach in [35]. In his thesis, Cook defines a technique that uses a combination of *proof planning* and *higher-order unification* [41]. Cook implemented his approach in the theorem prover λ Clam, and discovers higher-order functions in ML code that could then be parallelised using the parallelising compiler designed by Scaife et al. [107]. Cook's approach focusses on both the range and permutations of discoverable patterns; employing a backtracking algorithm, the technique is able to find multiple configurations of recursion schemes for each function inspected. Demonstrations of the approach include finding `map`, `foldr`, `foldl`, and `scanl` instances, with implementations defined for lists. By implementing his approach in a theorem prover, each of Cook's transformations

carry inherent proofs of functional equivalence. Cook's implemented approach requires translation from ML into λ Clam and vice versa; is defined for recursion schemes over lists only; and only considers simple induction over a single argument. The approach cannot discover coinductive, e.g. `zip`, or corecursive structures, e.g. `unfold`. It is likely to be non-trivial to extend this approach to other patterns, both including coinductive and corecursive patterns. An additional complication of this approach originates from the use of higher-order unification. In general, higher-order unification is an undecidable problem [66]. Semi-decidable formulations of higher-order unification have been developed [44], and have proven effective in a number of applications [65], but impose restrictions on the structure of functions that can be analysed. This is in addition to the inherent restriction that the inspected function must have a very similar structure to the pattern being searched for.

A more recent implementation of the higher-order unification approach is presented by Kannan in his thesis [74]. Kannan is primarily concerned with automatically introducing parallelism in functional programs, and so pattern discovery is presented as only a minor part of the approach. Indeed, Kannan's approach to pattern discovery is a combination of Distillation [55] and higher-order unification. Despite enabling a wider range of analysable functions, and as described in Section 3.1, the use of Distillation can *remove* patterns and therefore potentially reduce opportunities for parallelism [36, 41]. Here, higher-order unification is used to compare ASTs of Distilled functions with ASTs of skeletons. The range of skeletons that are introduced by this approach is also limited, and is difficult to extend. The skeletons provided are defined for traversals over lists only, and therefore operations that traverse non-list data-structures must be first translated, thereby introducing overheads.

3.3 Summary

With the development of pattern-based approaches to parallelism, recent decades have seen a lot of work on automatic approaches to pattern discovery. The majority of these approaches have focussed on the discovery,

introduction, and parallelisation of list homomorphisms *via* program calculational approaches. These approaches are generally limited by the range of functions they can analyse, i.e. functions that traverse a single list, and by the pattern they introduce. Functions that traverse non-list data types must be converted to lists, and almost homomorphisms must be converted to list homomorphisms using a projection function; both introduce runtime overheads. Finally, extending the range of patterns that are discoverable is non-trivial and takes significant amounts of programmer time and effort. Other program calculational approaches have focussed on the discovery of hylomorphisms in code. Since hylomorphisms have received less interest, the range of functions that can be rewritten as hylomorphisms is very limited. Indeed, near hylomorphisms cannot be found using current techniques. Non-calculational approaches to pattern discovery are also limited. Heuristics-based approaches are difficult to extend, and highly specific to languages. Program slicing is introduced as a means of automatic pattern discovery, and while it enables the inspection of individual expressions instead of whole functions, it requires analysed functions to have a particular structure, and expanding the range of patterns it can discover is non-trivial. Finally, approaches that use higher-order unification are, in principle, able to discover a wide range of patterns, including specialisations of more general patterns, but cannot discover simultaneously inductive and corecursive patterns. Moreover, these approaches require that extensive normalisation is performed prior to comparing the structures of the inspected function and a given pattern. This is effectively a restriction on the range of analysable patterns.

Current approaches to automatic pattern discovery are therefore primarily limited by the range of functions that they can analyse, and by the range of patterns that they can detect. While the range of functions that are analysable can be mitigated, often this results in other limitations or introduced overheads. The limited range of patterns is a greater restriction, where most current approaches can find only one generic pattern, or a few specific patterns. No current approaches are easily extensible to the point where a programmer can discover arbitrary patterns in a program.

Automatic Pattern Discovery via Program Slicing

In this chapter we describe a novel approach to discover computations in recursive functions that can be performed as part of a fixpoint function of a call to `map`. Explicit calls to `map` are effective *loci* of potential parallelism, provided that all mapped computations are independent [12]. *Program slicing* [117] can be used to statically inspect how the values of variables change between recursive calls in a function. Furthermore, by inspecting how values of variables change between recursive calls, it is possible to determine which computations occur independently, and so can be lifted into a `map` operation.

Section 4.1 gives an overview of the approach using a simple example to illustrate each stage. Section 4.2 defines a simple expression language that we use as the basis of our analysis, and describes the assumptions that are required for the technique to work. Section 4.3 presents the method for classifying computations in recursive functions that can then be potentially lifted into a `map` operation. In Section 4.3.2, we define a novel program slicing algorithm to aid classification of variables and operations. Section 4.4 describes the implementation of our approach in Erlang. Section 4.5 defines a composite refactoring that enables automatic rewriting of inspected functions. Section 4.6 describes our experiments and results. Finally, in Section 4.7, we discuss the strengths and weaknesses of the presented approach.

The core content of this chapter was published in the 2017 Workshop on Functional High-Performance Computing (FHPC) [12].

4.1 Illustrative Example

This chapter assumes the existence of a *parallel map* skeleton, such as those presented in Section 2.2, that can be swap-replaced for the standard sequential `map` operation over lists. Multiple parallelisations are possible; e.g. `parList` from the Haskell Strategies library [111], `parMap` from the Par Monad [87], or using GPUs via the Accelerate library [33]. The *parallel map* skeleton can be used to safely plug-replace any standard use of `map`, *provided that all mapped computations are independent*.

Consider a simple Haskell Sudoku solver, `sudoku`, that solves a list of Sudoku puzzles [86].

```
1 sudoku [] = []
2 sudoku (p:ps) = solve p : sudoku ps
```

Here, `sudoku` is implicitly sequential: using the function `solve`, it solves each puzzle, `p`, in turn. It also contains no `map` operations that can be used as *loci* of potential parallelism. However, the function application expression, `(solve p)`, could, in fact, be performed *independently* on each element of the input list, `(p:ps)`. The application of `solve` might therefore be *lifted*, perhaps by *lambda lifting* [73], outside the body of `sudoku`. Its result can then be passed as an additional argument to the helper function, `sudoku'`, derived as part of the lambda lifting.

```
1 sudoku ps =
2   sudoku' ps (map solve ps)
3   where
4   sudoku' [] [] = []
5   sudoku' (p:ps) (q:qs) = q : sudoku' ps qs
```

Since `sudoku'` reconstructs the result of the introduced `map` operation, `sudoku` can be further simplified, perhaps by refactoring, to remove `sudoku'`:

```
1 sudoku ps = map solve ps
```

It is then easy to introduce parallelism to `sudoku` by replacing the newly-introduced `map` operation with a parallel equivalent, e.g. `parMap` from the `Par Monad` [87].

```
1 sudoku ps = runPar $ parMap solve ps
```

Here, `runPar` runs a parallel computation and returns its result, and `parMap` spawns a child process for each input.

As in the above example, the approach described in this chapter inspects application subexpressions that are not recursive calls, hereafter *operations*, to see if they can be performed as part of a fixpoint function passed to a `map`. In the above example, `sudoku` had one operation, `(solve p)`, and that operation could be lifted into a `map`. Whether an operation might be lifted out of a recursive function and then parallelised depends solely on: *i*) function arguments occurring as subexpressions in the operation; and *ii*) how the values of arguments to the recursive function differ between the stages of recursion. In `sudoku`, for example, the list argument reduces by one element with each recursive call. This can be considered a form of *dependency analysis*, a topic that has been heavily studied. For example, control- and data-flow analyses [4] are well-known techniques for calculating dependencies between individual statements of a program, and can be used to produce a Program Dependence Graph, or PDG. Applying standard control- and data-flow analyses to `sudoku` might produce a graph similar to Figure 4.1. This graph does not immediately reveal how the values of variables change between recursive calls, or which operations can be lifted.

Program slicing is a technique that is used to extract all the program statements that *may influence*, or *may be influenced by*, a given statement from the same program, the *slicing criterion* [106]. Whilst there have been many variants of program slicing since Weiser first introduced the concept in 1981 [117], generally, a (static) slice is calculated using a PDG. Once the PDG of the given program has been calculated, the slice itself is produced by taking the vertex representing the criterion and, depending on the specific algorithm, following the edges that lead to it, that lead

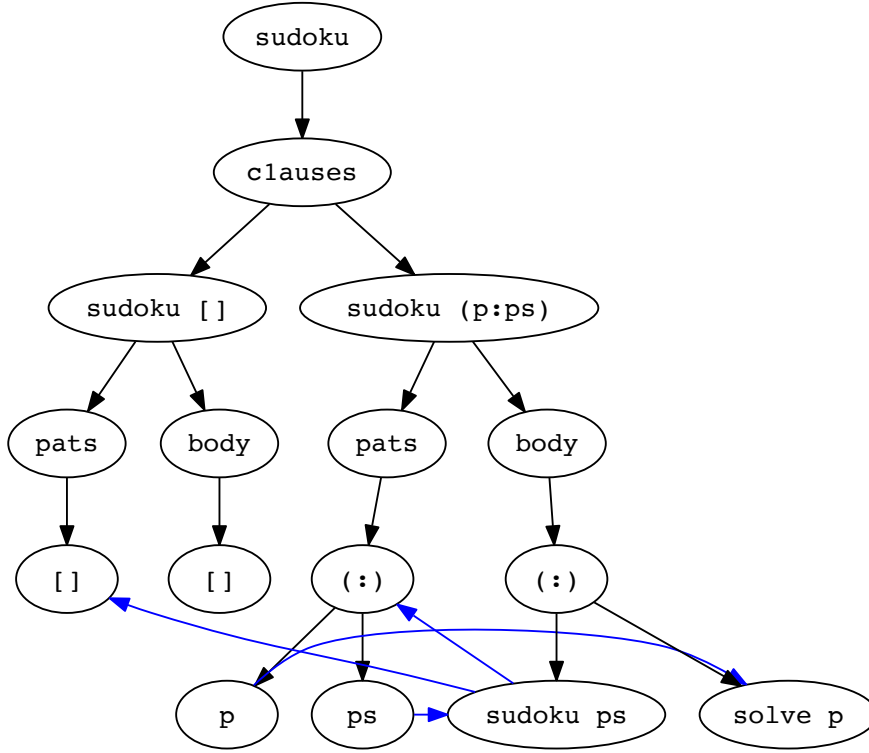


Figure 4.1: Program Dependence Graph for `sudoku`; control-flow in black, data-flow in blue.

from it, or perhaps both. The slice can then be presented in a number of ways, most commonly as the program where all statements that are *not in the slice* are removed. For example, the slice of `sudoku`, with criterion `p`, is:

```

1 sudoku _ = undefined
2 sudoku (p:_) = solve p : undefined

```

Figure 4.2 shows the PDG of `sudoku` with the node representing the criterion, and all nodes reachable by travelling *up* control-flow arrows and *down* data-flow arrows from the criterion, highlighted in red. The sliced definition of `sudoku` includes those nodes in the slice; any node that is *not* in the slice is represented by the wildcard pattern, `(_)`, or by `undefined`, indicating its removal from the definition.

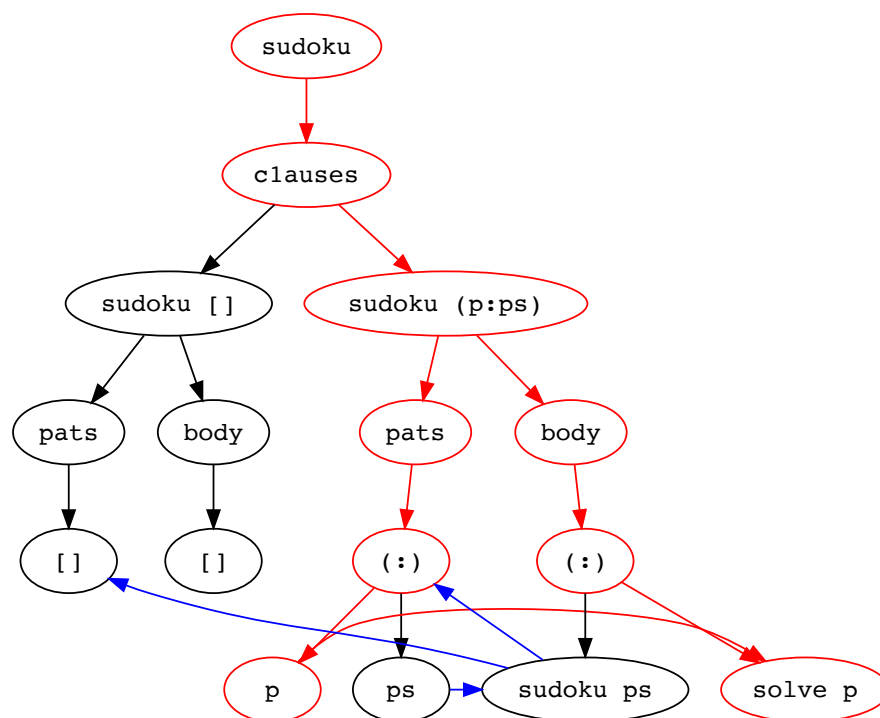


Figure 4.2: Program Dependence Graph for `sudoku`; control-flow in black, data-flow in blue; slice with criterion `p` in red.

Slicing might therefore be thought of as a *narrowing of focus* on only the slicing criteria, the any related expressions, and with all the irrelevant parts stripped away. For the purposes of this chapter, however, in the interest of determining how arguments are *used* and *updated*, the slice can be narrowed further, and so make the decision of whether an operation may be lifted into a `map` operation easier. A slice only needs to be a set of variables, where inclusion in the slice indicates *use*, and annotated inclusion in the slice indicates *update*. Taking a slice of `sudoku`, with criterion `p`, now produces the slice $\{p\}$, indicating its *use* as an argument to `solve`, and non-update in the recursive call; i.e. that same `p` is not accessed in any successive recursive calls. A slicing criterion can be a set of expressions; e.g. a slice of `sudoku` with its *first argument*, i.e. `[]` in Line 1 and `(p:ps)` in Line 2, as a slicing criterion produces: $\{p, ps\}$.

Formal definitions of both *usage* and *update* classifications are given in Section 4.3; but we provide an intuition here. To determine how arguments are *used* and *updated*, function arguments are used as slicing criteria. A slice is taken for each argument. A variable, v_i , where v_i is the i^{th} function argument declared, is *used* if it appears as a subexpression in the definition of the inspected recursive function, except in the case where that variable is passed as the i^{th} argument to a recursive call. For example, given the definition of `elem`, which returns `True` if some `a` is an element in a list,

```
1 elem a [] = False
2 elem a (x:xs) = if a == x then True else elem a xs
```

`a`, the first declared argument, is *used* because it exists as an argument to the infix equality check. Conversely, without the infix equality check,

```
1 elem a [] = False
2 elem a (x:xs) = elem a xs
```

`a` is *not* considered *used*. The right-hand side of the equation in Line 2 comprises a recursive call, and `a` is passed as the first argument. Unless `a` is to be *updated*, `a` *must* be passed to all recursive calls to preserve its value.

Although variables in a functional language such as Haskell are immutable, and each function application binds new values to each of its arguments, the differences in the value of a specific argument between successive recursive calls are deemed changes to the corresponding bound variables. A change to the value of the argument v_i , is considered to be *significant*, and therefore v_i is considered to be *updated*, when the i^{th} argument of some recursive call is *not*:

1. v_i itself, or a variable declared by case-splitting v_i ;
2. syntactically equivalent to v_i ; nor
3. a cons-expression that prepends an expression to v_i (or its syntactic equivalent).

In `sudoku`, for example, the recursive call takes the tail of the list argument, i.e. `ps`, which is declared by (implicitly) case-splitting on the list argument. Consequently the list argument is *not* considered to be *updated*. Similarly, the list argument to `elem` is not considered *updated* for the same reason. Finally, `a`, the first argument declared in `elem`, is also not considered to be *updated* because the first argument to the recursive call is `a` itself. Finally, consider the definition of `sum`, which accumulatively sums a given list:

```
1 sum x [] = x
2 sum x (y:ys) = sum (x+y) ys
```

Here, `x` is considered to be *updated* since the first argument to the recursive call in Line 2 is not: *i*) `x` itself or a variable declared by case-splitting `x`; *ii*) syntactically equivalent to `x`; or *iii*) a cons-expression that prepends an expression to `x`. The slice of `sum`, with criterion `x`, is: $\{x, \bar{x}\}$, where \bar{x} denotes that `x` is *updated*.

If an argument is *used* and not *updated*, as for all arguments in `sudoku` and `elem`, it follows that the argument can be referred to in some `map` operation independently of each input. Similarly, if the argument is *updated but not used*, it can be *updated* in some `map` operation independently of each input. For example, when constructing state in a monad. In either of these cases, the argument is considered *clean*. However, when an argument is both *used and updated*, it indicates that the *usage* and *update* of the variable is not independent of the other stages of the recursion. For example, in `sum`, `x` is considered to be both *used* and *updated*. This indicates that `x` depends upon at least one other element in the list to produce a new value, which is propagated through the recursion. Any function using such a variable, e.g. `(+)` in `sum`, cannot be safely lifted into a `map` operation, and that variable is considered to be *tainted*.

Having determined for each argument in a recursive function whether they are *clean* or *tainted*, each operation within that same function definition can be inspected to see whether it can be safely lifted into a `map` operation. An operation can be lifted into a `map` operation, deemed *unobstructive*, when *none* of its arguments contain as a subexpression a *tainted* variable or a recursive call. Otherwise, an operation is considered

to be *obstructive*. For example, `sum` has only one operation, $(x+y)$, which takes one *tainted* variable, i.e. `x`, and is therefore classified as *obstructive*. `elem` has two operations, $(a == x)$ and the if-expression, which we treat as syntactic sugar for a function. Here, $(a == x)$ is classified as *unobstructive* as both `a` and `x` are *clean*. Conversely, the if-operation is classified as *obstructive* due to the recursive call in its `False` branch. Finally, `sudoku` has one operation, $(\text{solve } p)$, which is considered to be *unobstructive*, since `p` is classified as *clean*. *Unobstructive* operations can potentially be lifted into `map` operations, and by extension are candidates for parallelisation. Recall that for `sudoku`, as $(\text{solve } p)$ is *unobstructive*, it can be lifted into a `map`, and `sudoku` rewritten and parallelised as before:

```
1 sudoku ps = runPar $ parMap solve ps
```

4.2 Preliminaries and Assumptions

We illustrate our approach over the simple expression language, E .

$$\begin{array}{lcl}
 e \in E & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \mathbb{Z} \\
 & | & \text{var} \\
 & | & \text{nil}_\tau \\
 & | & \text{cons}_\tau e e \\
 & | & \text{case } \text{var} \text{ of } \text{nil}_\tau \rightarrow e, \text{cons}_\tau \text{var } \text{var} \rightarrow e \\
 & | & e \vec{e} \\
 & | & \lambda \vec{v} \vec{a} r \rightarrow e \\
 & | & \text{fix } e
 \end{array}$$

E is a simple, strict, functional language. Its terms for a common subset of functional languages: boolean constants, `true` and `false`; integer constants, $z \in \mathbb{Z}$; variables, `var`; list constructors, `nilτ` and `consτ e e`; case discrimination on lists, `case var of nilτ → e, consτ var var → e`; function application, `e \vec{e}` ; lambda expressions, `λ $\vec{v} \vec{a} r \rightarrow e$` ; and fixpoints, `fix e`. Constructors are restricted to `cons`-lists for simplicity and clarity of presentation, but

$$\begin{array}{c}
 \text{BOOL}_1 \frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad \text{BOOL}_2 \frac{}{\Gamma \vdash \text{false} : \text{bool}} \\
 \\
 \text{INT} \frac{}{\Gamma \vdash \mathbb{Z} : \text{int}} \quad \text{VAR} \frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau} \\
 \\
 \text{LIST}_1 \frac{}{\Gamma \vdash \text{nil}_\emptyset : \text{list } \tau} \quad \text{LIST}_2 \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \text{list } \tau}{\Gamma \vdash \text{cons}_\emptyset e_1 e_2 : \text{list } \tau} \\
 \\
 \text{CASE} \frac{\Gamma \vdash xs : \text{list } \tau_1 \quad \Gamma \vdash y : \tau_1 \quad \Gamma \vdash ys : \text{list } \tau_1 \quad \Gamma \vdash e_1 : \tau_2 \quad \Gamma \cup \{y : \tau_1, ys : \text{list } \tau_1\} \vdash e_2 : \tau_2}{\Gamma \vdash \text{case } xs \text{ of } \text{nil}_{\tau_1} \rightarrow e_1, \text{cons}_{\tau_1}(y, ys) \rightarrow e_2 : \tau_2} \\
 \\
 \text{APP} \frac{\Gamma \vdash e_0 : \vec{\tau} \rightarrow \tau_m \quad \Gamma \vdash \vec{e} : \vec{\tau}}{\Gamma \vdash e_0 \vec{e} : \tau_m} \quad \text{FUN} \frac{\Gamma \cup \{\vec{x} : \vec{\tau}\} \vdash e : \tau_m}{\Gamma \vdash \lambda \vec{x} \rightarrow e : \vec{\tau} \rightarrow \tau_m} \\
 \\
 \text{FIX} \frac{\Gamma \vdash e : (((\tau_2, \dots, \tau_n) \rightarrow \tau_m), \tau_2, \dots, \tau_n) \rightarrow \tau_m}{\Gamma \vdash \text{fix } e : (\tau_2, \dots, \tau_n) \rightarrow \tau_m}
 \end{array}$$

 Figure 4.3: Typing judgements for E , determining simple types in T .

the approach is extensible to other types, given a definition of *variable update* (Definition 4.3.1) for that type. Similarly, the approach can be extended to arbitrary types, given that a definition of *variable update* can be derived for arbitrary constructors. Vector notation, e.g. \vec{e} , refers to a non-empty tuple: $\vec{e} \equiv (e_1, \dots, e_n), n \geq 1$. In order to simplify our presentation, list constructors and case discriminators are annotated with the (monomorphic) type of the list elements, τ . The corresponding type language, T , is shown below.

$$\begin{array}{l}
 \tau \in T ::= \text{bool} \\
 \quad | \text{int} \\
 \quad | \text{list } \tau \\
 \quad | \vec{\tau} \rightarrow \tau
 \end{array}$$

The typing judgements in Fig. 4.3 then determine the *well-formedness* of expressions in E with regard to their monomorphic types in T . A statement, $s \in S$, is an assignment.

$$\begin{array}{l}
 s \in S ::= \mathbf{def} \text{ var} = \lambda \vec{v} \vec{a} r \rightarrow e \\
 \quad | \mathbf{def} \text{ var} = \text{fix } e
 \end{array}$$

Statements appear only at the top level of a program. A variable may be bound either to a lambda expression or to a fixpoint expression. Those bindings are then in scope for the duration of program. A program, $p \in P$, is a series of statements.

$$p \in P ::= s \\ | s ; p$$

P can be thought of as an intermediate representation to which, e.g., Haskell or Erlang are compiled, similar to Core Haskell. For example, the Haskell definition of `sudoku`,

```
1 sudoku [] = []
2 sudoku (p:ps) = solve p : sudoku ps
```

can be translated into a term in P :

```
def sudoku = fix λ(f,x) →
  case x of
    nilπ →
      nilπ,
    consπ(y, ys) →
      consπ(solve(y), f(ys))
```

This chapter will use Haskell syntax for examples in order to improve readability. All examples given can be translated into P following a similar principle to the above example. Our approach will inspect only the code provided and does not presume to predict possible compiler optimisations, e.g. fusions [48] or worker-wrapper [49] transformations. As an intermediate representation, these techniques can be applied after, or prior to, the application of our approach.

Where pattern matching is used, we will use as-patterns to indicate the implicit list variables; e.g.

```
1 sudoku ps₀@[] = []
2 sudoku ps₀@(p:ps) = solve p : sudoku ps
```

For clarity, all the variables in our examples will be consistent across function clauses. All variables are assumed to be unique under α -conversion,

at both the statement and expression levels. Type environments, Γ , are defined to be a set of bindings of variables to types:

$$\Gamma \in \{var : T\}$$

As usual, all values in the domain of Γ are assumed unique, and $\Gamma(x)$ denotes the τ of x in Γ , such that $\exists \tau \in T, (x : \tau) \in \Gamma$. For a given program p , the *program environment*, Γ_p , contains all the variables in p .

Definition 4.2.1 (Program Environment, Γ_p). *Given some program p and the set of variables $X \subseteq \text{var}$ that occur (either free or bound) in p , we define an environment Γ_p to be a set such that $\forall x \in X, \exists \tau \in T, \Gamma_p = \{x : T\} \cup \Gamma_p$.*

The above `sudoku` definition, for example, has the Γ_p :

$$\Gamma_p = \{x : \text{list}(\text{list int}), y : \text{list int}, ys : \text{list}(\text{list int}), \\ f : (\text{list}(\text{list int})) \rightarrow (\text{list}(\text{list int})), \text{solve} : (\text{list int}) \rightarrow (\text{list int}), \dots\}$$

We omit the variables and types of `solve` for clarity. For the rest of this chapter, we will assume that for all given variables $x \in \Gamma_p$.

It is useful to define the notion of *subexpressions* in E , since subexpressions are a key element in both slicing and classification definitions.

Definition 4.2.2 (Subexpression). *Given two expressions e, e' , say that e' is a subexpression of e (denoted $e' \ll e$) when*

$$\frac{e' = e}{e' \ll e} \quad \frac{e' \ll e_1 \vee e' \ll e_2}{e' \ll \text{cons}_\tau e_1 e_2}$$

$$\frac{e' \ll e_1 \vee e' \ll e_2}{e' \ll \text{case } x \text{ of } \text{nil}_\tau \rightarrow e_1, \text{cons}_\tau x' x'' \rightarrow e_2}$$

$$\frac{\exists i \in [0, n], e' \ll e_i}{e' \ll e_0 \vec{e}} \quad \frac{e' \ll e}{e' \ll \lambda \vec{x} \rightarrow e} \quad \frac{e' \ll e}{e' \ll \text{fix } e}$$

Subexpressions form a partial order relation.

For example, a function that adds 1 to its argument, $\lambda(x) \rightarrow \text{add}(x, 1)$, has five subexpressions: *i)* 1, *ii)* x , *iii)* sum , *iv)* $\text{sum}(x, 1)$, and finally

$v) \lambda(x) \rightarrow \text{sum}(x, 1)$. We will refer to any application that is a subexpression of a fixpoint and that is not a recursive call as an *operation*. For example, the fixpoint expression

$$\begin{aligned} \text{fix } \lambda(f, xs) \rightarrow \\ \text{case } xs \text{ of} \\ \text{nil}_\pi \rightarrow \\ 0, \\ \text{cons}_\pi(y, ys) \rightarrow \\ \text{plus}(y, f(ys)) \end{aligned}$$

which sums the elements of a list of integers, has one operation: *plus*. $f(ys)$ is *not* an operation because it is a recursive call.

Functions are introduced using a λ -expression. They are always pure, are uncurried, and are never partially applied. They may, however, be higher-order. Recursive (function) definitions are always introduced using an explicit fixpoint expression, as in e.g.:

$$\text{fix}(\lambda(f, xs) \rightarrow f(xs)).$$

The form of recursion is not otherwise restricted; general recursive forms are allowed, for example.

Lists are defined to be an ordered collection of elements, where those elements are accessed *via* case-expressions. As shown by the typing rules of Figure 43, case discrimination is restricted to lists of some type τ . We assume the existence of built-in functions (e.g. *if*, *eq*, *plus*) for discrimination and operations on integers and booleans. Case-expressions can be extended to other types, given an additional check on the type of the discriminated variable in the relevant definitions. We limit case-expressions here to simplify our presentation. In the non-nil branch, new variables are bound respectively to the first element in the list (i.e. the head) and to the remainder of the list (i.e. the tail). A corresponding Reachability Relation is defined for each case-expression.

Definition 4.2.3 (Reachability Relation). *Given a program p , a program environment Γ_p , and a case-expression in p , $e = \text{case } xs_0 \text{ of } \text{nil}_\tau \rightarrow e_1, \text{cons}_\tau x \text{ } xs \rightarrow e_2$, we say that $x \triangleleft_p xs_0$ and $xs \triangleleft_p xs_0$. The transitive closure of the reachability*

relation is defined such that $z \triangleleft_p^+ xs_0$ when $\exists y, z \triangleleft_p^+ y \wedge y \triangleleft_p xs_0$. The reflexive-transitive closure of the reachability relation is defined such that $y \triangleleft_p^* xs_0$ when $\exists y, y \triangleleft_p^+ xs_0 \vee y = xs_0$.

For example, $p \triangleleft_p^* ps_0$, $ps \triangleleft_p^* ps_0$, and $ps_0 \triangleleft_p^* ps_0$ all hold for the case-expression in `sudoku`. The Reachability Relation will be used to calculate the *program slice* for a given expression and variable. It is also useful to know when a *cons*-expression reconstructs xs_0 (e.g. $(p:ps)$ in `sudoku`), so that xs_0 can be included in the slice. The syntactic equivalence relation for lists is used to detect such list reconstructions.

Definition 4.2.4 (Syntactic Equivalence for Lists). *Given some program p ; argument x of type list τ , and an expression, where that expression is either i) the special symbol ε , denoting an empty expression and that an expression for which it has been substituted is syntactically equivalent to x by Definition 4.3.2; or ii) a cons-expression $e = \text{cons}_\tau e' e''$. We say that e is syntactically equivalent to x , denoted $e \equiv x$, when $e = \varepsilon$ or $e = \text{cons}_\tau e' e''$ given that $\exists y, \exists ys, e' = y \wedge e'' \equiv ys \wedge y \triangleleft_p x \wedge ys \triangleleft_p x$.*

For example, where ps_0 is case-split in `sudoku` into p and ps , ps_0 is syntactically equivalent to a *cons*-expression with x as the first argument and ps as the second argument; i.e. $ps_0 \equiv \text{cons}_\tau p ps$.

4.3 Determining Obstructiveness

Each operation is inspected to determine whether that operation is *obstructive*. An *obstructive* operation is an operation that has either: i) any arguments whose subexpressions contain a recursive call; or ii) any arguments that are both used in the body of the fixpoint and whose value is changed *significantly* in any recursive call. All other operations are classified as *unobstructive*. So, for example, $(\text{solve } p)$ in `sudoku` is classified as *unobstructive* because `solve` is a unary function that takes only the head of the list. Conversely, the $(x+y)$ operation in `sum` is classified as *obstructive* because it takes the first argument, x , which is *updated* between recursive calls. We define both obstructiveness and unobstructiveness formally in Definition 4.3.6. The nature of *significance* is defined below.

4.3.1 Variable Usage and Update

From the intuitive definition of obstructiveness above, all variables in Γ_p for some fixpoint expression, $e = \text{fix } \lambda(f, x_1, \dots, x_n) \rightarrow e'$, must be first inspected to classify operations in e as (un)obstructive. Each variable $x \in \Gamma_p$ is classified as *used*, *updated*, or both, in e .

Although all the variables in P are immutable, and each function application binds new values to each of its arguments, the value of some variable x is considered (potentially) *changed* in e when x is the i^{th} argument to f and there exists some recursive call to f in e' or any of its subexpressions. That is, the differences in the value of a specific argument between successive recursive calls are considered to be changes to the corresponding bound variables. We extend this notion with the concept of *significantly* changed. Intuitively, a change in value is considered to be *significant* when for some recursive call, $f(e_1, \dots, e_n)$, the i^{th} argument, e_i , is not: *i*) a variable that is reachable from x ; *ii*) syntactically equivalent to x ; nor *iii*) a cons-expression that prepends an expression to x (or its syntactic equivalent) that contains no subexpression that is either a recursive call or a variable reachable from x . Variables whose value is significantly changed in this way are classified *updated*.

Definition 4.3.1 (Variable Update). *Given a program p , a program environment Γ_p , a fixpoint expression $e = \text{fix } (\lambda(f, x_1, \dots, x_n) \rightarrow e')$ in p , and an argument x , where x is the i^{th} argument to f , x is considered to be updated in e (denoted $\bar{x} \sim_x e$) when there exists a recursive call, $f(e_1, \dots, e_n) \ll e'$ such that $\neg(e_i \triangleleft_p^* x) \wedge (e_i \not\equiv x) \wedge \neg(e_i = \text{cons}_\tau e_k e_l \wedge e_l \equiv x \wedge \neg(\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn})))$*

Usage is a simpler concept: a variable is considered to be *used* when it occurs as a subexpression of a fixpoint expression. There is one exception to this: when a subexpression is the i^{th} argument to a recursive call then it is not considered to be *used*. We first define the notion of *variable-usage escapement*. This is to avoid the *used* classification of x , and ultimately the membership of x in the slice, when e_i is an x itself or when prepending to x . For example, it avoids the erroneous classification of x in:

```

1  f a x = if a < 5
2          then f (a+1) x
3          else f (a-1) a
    
```

Here, in Line 3, x is correctly classified as being *updated*. In Line 2, x is not updated, but it should also not be considered to be *used*, since x is necessary as an argument to the recursive call in order to retain the value of x .

Definition 4.3.2 (Variable-Usage Escapement). *For some fixpoint expression $e = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow e')$, $\forall f(e_1, \dots, e_n) \ll e'$, and for some $i \in [1, n]$, given x , the i^{th} argument to f , we substitute the special symbol ε for e_i whenever $e_i = x$ or $e_i \equiv x$. When $e_i = \text{cons}_\tau e_k e_l \wedge e_l \equiv x \wedge \neg(\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn})))$ holds, we substitute ε for e_l . Variable-usage escapement of e with respect to x is denoted $e \setminus_\varepsilon x$.*

In the above definition of f , for example, the x in Line 2 would be substituted for ε .

```

1  f a x = if a < 5
2          then f (a+1) ε
3          else f (a-1) a
    
```

We can now define variable *usage*.

Definition 4.3.3 (Variable Usage). *Given some fixpoint expression, e , where $e = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow e')$, in a program p , and a variable x where x is the i^{th} argument to f , a variable y is considered to be used in e (denoted $y \sim_x e$) when $y \triangleleft_p^* x$ and y exists as a subexpression to the variable-usage escaped e ; i.e. $y \ll (e \setminus_\varepsilon x)$. When there exists an expression, such that $e_1 \ll (e \setminus_\varepsilon x) \wedge e_1 \equiv x$, $x \sim_x e$ holds.*

To illustrate this, consider the following functions.

```

1  f a xs0@[] = a
2  f a xs0@(x:xs) = f x xs
3
4  g xs0@(x:xs) ys0@(y:ys) = g (x:xs) (x:ys)
5  g xs0 ys0 = xs0
    
```

```

6
7 h xs0@(x:xs) ys0@(y:ys) = h (y:(x:xs)) (y:(y:ys))
8 h xs0 ys0 = xs0

```

For f , the value of a is *updated* since $\neg(x \triangleleft_p^* a)$. However, the list argument is *not updated* since xs is reachable from xs_0 ; i.e. $xs \triangleleft_p^* xs_0$. All variables, excluding xs_0 , in the program environment of f are considered to be *used*: a is used in Line 1; x in the first argument of the recursive call; and xs in the second argument of the recursive call. For g , xs_0 is *not updated* since $(x:xs)$ is syntactically equivalent to xs_0 ; i.e. $\text{cons}_\tau x xs \equiv xs_0$. However, ys_0 is *updated* since $(x:ys)$ is not syntactically equivalent to ys_0 . All variables apart from y and xs are considered to be *used* in g : xs_0 is used both in Line 6 and as the semantically equivalent first argument to the recursive call; x and ys are used in the second argument of the recursive call. Finally, for h , xs_0 is *not updated*, since y (which is not reachable from xs_0) is prepended to xs_0 . However, ys_0 is *updated* since y is prepended to ys_0 and y is reachable from ys_0 . y , xs_0 , and ys_0 are all considered to be *used* in h : xs_0 in Line 9 and the *cons* in the first argument of the recursive call; ys_0 in the *cons* in the second argument of the recursive call; y in the *cons* of both arguments of the recursive call.

4.3.2 Slicing Algorithm

Intuitively, a slice $\Sigma^{e|x}$ of an expression e with criterion x is a set of variables that indicate whether a variable y is *used* in e and whether x is *updated* in e , denoted by the annotation \bar{x} . A slice can be used to categorise variables, and ultimately to determine the obstructiveness of operations.

Definition 4.3.4 (Slice). *Given some program p , the program environment Γ_p , a fixpoint expression $e = \text{fix } \lambda(f, x_1, \dots, x_n) \rightarrow e'$, and a variable x , where x is the i^{th} argument to f , we say that the slice of e with criterion x , denoted $\Sigma^{e|x}$, is the set of variables such that $\forall y \in \Sigma^{e|x}, (y \sim_x e) \vee (y = \bar{x} \wedge \bar{x} \sim_x e)$.*

To illustrate the slicing relation, recall the definition of `sudoku`

```

1 sudoku ps0@[] = []
2 sudoku ps0@(p:ps) = solve p : sudoku ps

```

Considering each variable in `sudoku`: both $p \in \Sigma^{\text{sudoku}|ps_0}$ and $ps \in \Sigma^{\text{sudoku}|ps_0}$ hold since both p and ps are reachable from ps_0 (i.e. $p \triangleleft_p^* ps_0$ and $ps \triangleleft_p^* ps_0$) and both p and ps are *used* in Line 2; conversely, *neither* $ps_0 \in \Sigma^{\text{sudoku}|ps_0}$ *nor* $p\bar{s}_0 \in \Sigma^{\text{sudoku}|ps_0}$ hold, since ps_0 is not considered to be *used*, and ps_0 is not considered to be *updated*, in `sudoku`. For the definition of `sum`:

```

1  sum x ys@[] = x
2  sum x ys@(y:ys) = sum (x+y) ys

```

$x \in \Sigma^{\text{sum}|x}$ holds since x is *used* in both Lines 1 and 2. $\bar{x} \in \Sigma^{\text{sum}|x}$ also holds since x is *updated* in the recursive call in Line 2 by the $(x+y)$ operation. As with `sudoku`, both $y \in \Sigma^{\text{sum}|ys_0}$ and $ys \in \Sigma^{\text{sum}|ys_0}$ hold since both y and ys are considered to be *used* in Line 2. Conversely, *neither* $ys_0 \in \Sigma^{\text{sum}|ys_0}$ *nor* $y\bar{s}_0$ hold since ys_0 is *not* considered to be *used* or *updated* in `sum`. The slices of `sudoku` and `sum` are:

$$\begin{aligned}
 \Sigma^{\text{sudoku}|xs_0} &= \{x, xs\} \\
 \Sigma^{\text{sum}|x} &= \{x, \bar{x}\} \\
 \Sigma^{\text{sum}|ys_0} &= \{y, ys\}
 \end{aligned}$$

The slice $\Sigma^{e|x}$ is calculated using the inference rules from Figure 4.4. For clarity of notation, we will write $\Sigma^{e|x}$ as Σ since neither e nor x can be changed within a slicing operation. Judgements are in the form:

$$\Gamma, f, x, i \vdash e : \Sigma$$

where Γ_p is the environment for some program p ; f is the name of the fixpoint function being sliced; x is the slicing criterion that is declared as the i^{th} argument of f ; e is the expression in p that is being sliced; and Σ is the resulting slice. For literal expressions and variables that are not reachable from x , e produces an empty slice, represented by the rules `BOOL1`, `BOOL2`, `INT`, `VAR2`, and `LST1`. A variable that is reachable from x , as a usage of x , produces the slice containing that variable (`VAR1`). Cons-expressions that are syntactically equivalent to x produce the slice containing x , and the two subexpressions (`LST2`). All other expressions that are not recursive calls produce the union of the slices

of their subexpressions, as stated in the rules LST_3 , $CASE$, APP , FUN , and FIX . Finally, $REC-APP$ determines whether x is updated in a recursive call, producing the appropriate slice. $REC-APP$ has two main premises that: *i*) slice all subexpressions that are passed to f , apart from the i^{th} argument; and *ii*) determines whether x is updated. The second premise is a disjunction of four implications: *i*) when e_i is either x itself, or is syntactically equivalent to x ; *ii*) when e_i is a variable that is defined via case-discrimination on x ; *iii*) when x is prepended by some expression e_k that does not contain either a recursive call or a subexpression that is a variable that is *reachable* from x ; or *iv*) when x is considered *updated*. In the first case, the slice, Σ_i , for the i^{th} argument to f , e_i , is the empty set since the argument preserves the value of x for the next recursive call. In the second case, Σ_i is the singleton set containing e_i itself since the value of x is changed, and a case-derived variable is *used*, but x is not considered to be *updated*. In the third case, Σ_i is the slice of the subexpression that is prepended to x . Finally, in the fourth case, Σ_i is the slice of e_i and x is considered to be *updated*.

Theorem 4.1 (Soundness of Slicing Algorithm). *For all programs, p , and their respective program environments, Γ_p ; for all fixpoint expressions, e , in p where,*

$$e = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow e')$$

and for all variables x , such that $x = x_\mu$ where $\mu \in [1, n]$, we can derive the slice of e with criterion x , $\Sigma^{e|x}$, by

$$\Gamma_p, f, x, \mu \vdash e : \Sigma^{e|x}$$

It follows that

$$\forall y, y \in \Sigma^{e|x} \text{ implies that } y \sim_x e$$

where $y \sim_x e$ denotes that y is either used or updated in e according to Definitions 4.3.3 and 4.3.1, respectively.

Intuitively, the above soundness property states that the slicing algorithm in Figure 4.4 is sound with respect to the slicing definition, Definition 4.3.4, such that our algorithm produces a slice whose members are only those variables that are considered to be *used* or *updated* in e

with respect to x . We give the proof for the above soundness property in Appendix A. We conjecture that slices are unique for each e and x . We defer the proof of uniqueness to future work.

4.3.3 Classifying Variables

Variables can be classified as: *global*, *clean*, or *tainted*. Global variables are those that are in scope in e , but which are declared and bound outside of e . They may be *used*, but cannot be *updated* during the evaluation of e ; they are therefore treated as literals. Variables that are classified as either *clean* or *tainted* are ones that are either defined in the fixpoint function (i.e. x_1, \dots, x_n), or in case-subexpressions of e' , where whenever $\forall v, w \in \Gamma_p$ such that $v \triangleleft_p^+ w$, it follows that v has the same classification as w .

Definition 4.3.5 (Variable Taint). *Given a program p , program environment Γ_p , a fixpoint expression $e = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow e')$ in p , a variable x , and a slice $\Sigma^{e|x}$ of e with criterion x , then x is classified as tainted when $\bar{x} \in \Sigma^{e|x} \wedge x \in \Sigma^{e|x}$. The variable is otherwise classified as clean.*

Recall the definitions of `sudoku`, `elem`, and `sum`:

```

1  sudoku xs₀@[]          = []
2  sudoku xs₀@(x:xs)      = solve x : sudoku xs
3
4  elem a xs₀@[]          = False
5  elem a xs₀@(x:xs)      =
6    if x == a
7    then True
8    else elem a xs
9
10 sum x ys₀@[]           = x
11 sum x ys₀@(y:ys)       = sum (x+y) ys

```

$$\begin{array}{c}
 \text{BOOL}_1 \frac{}{\Gamma, f, x, i \vdash \text{true} : \emptyset} \quad \text{BOOL}_2 \frac{}{\Gamma, f, x, i \vdash \text{false} : \emptyset} \quad \text{INT} \frac{}{\Gamma, f, x, i \vdash \mathbb{Z} : \emptyset} \\
 \\
 \text{VAR}_1 \frac{y \triangleleft_p^* x}{\Gamma, f, x, i \vdash y : \{y\}} \quad \text{VAR}_2 \frac{\neg(y \triangleleft_p^* x)}{\Gamma, f, x, i \vdash y : \emptyset} \\
 \\
 \text{LST}_1 \frac{}{\Gamma, f, x, i \vdash \text{nil} : \emptyset} \quad \text{LST}_2 \frac{\text{CONS}_\tau y \, ys \equiv x}{\Gamma, f, x, i \vdash \text{CONS}_\tau y \, ys : \{y, ys, x\}} \\
 \\
 \text{LST}_3 \frac{\Gamma, f, x, i \vdash e_1 : \Sigma_1 \quad \Gamma, f, x, i \vdash e_2 : \Sigma_2 \quad \text{CONS}_\tau e_1 \, e_2 \neq x}{\Gamma, f, x, i \vdash \text{CONS}_\tau e_1 \, e_2 : \Sigma_1 \cup \Sigma_2} \quad \text{CASE} \frac{\Gamma, f, x, i \vdash e_1 : \Sigma_1 \quad \Gamma, f, x, i \vdash e_2 : \Sigma_2}{\Gamma, f, x, i \vdash \text{case } ys \text{ of } \text{nil}_\tau \rightarrow e_1, \text{CONS}_\tau z \, zs \rightarrow e_2 : \Sigma_1 \cup \Sigma_2} \\
 \\
 \text{REC-APP} \frac{\begin{array}{c} \forall j \in [1, i] \cup (i, n], \Gamma, f, x, i \vdash e_j : \Sigma_j \\ ((e_i = x \vee e_i \equiv x) \rightarrow \Sigma_i = \emptyset) \vee ((e_i \triangleleft_p^+ x) \rightarrow \Sigma_i = \{e_i\}) \vee \\ ((\exists e_k, \exists e_l, e_i = \text{CONS}_\tau e_k \, e_l \wedge (\exists v, (v \ll e_k \wedge v \triangleleft_p^* x)) \wedge e_l \equiv x) \rightarrow \Sigma_i = \emptyset) \vee \\ ((\exists e_k, \exists e_l \equiv x \vee (e_i = \text{CONS}_\tau e_k \, e_l \wedge e_l \equiv x \wedge (\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn})))))) \wedge \Gamma, f, x, i \vdash e_i : \Sigma \rightarrow \Sigma_i = \{\bar{x}\} \cup \Sigma)) \end{array}}{\Gamma, f, x, i \vdash f(e_1, \dots, e_n) : \bigcup_{j \in [1, n]} \Sigma_j} \\
 \\
 \text{APP} \frac{\forall j \in [0, n], \Gamma, f, x, i \vdash e_j : \Sigma_j \quad e_0 \neq f}{\Gamma, f, x, i \vdash e_0(e_1, \dots, e_n) : \bigcup_{j=0}^n \Sigma_j} \quad \text{FUN} \frac{\Gamma, f, x, i \vdash e : \Sigma}{\Gamma, f, x, i \vdash \lambda(y) \rightarrow e : \Sigma} \quad \text{FIX} \frac{\Gamma, f, x, i \vdash e : \Sigma}{\Gamma, f, x, i \vdash \text{fix } e : \Sigma}
 \end{array}$$

 Figure 4.4: Inference rules to calculate the slice $\Sigma^{e|x}$ for an expression e with criterion x .

Given the slices of each fixpoint variable in `sudoku`, `elem`, and `sum`,

$$\begin{aligned}\Sigma^{\text{sudoku}|xs_0} &= \{x, xs\} \\ \Sigma^{\text{elem}|a} &= \{a\} \\ \Sigma^{\text{elem}|xs_0} &= \{x, xs\} \\ \Sigma^{\text{sum}|x} &= \{x, \bar{x}\} \\ \Sigma^{\text{sum}|ys_0} &= \{y, ys\}\end{aligned}$$

we can proceed to classify the variables for all three functions. In both `sudoku` and `elem`, xs_0 is classified as *clean*, since case-split derived variables y and ys are *used*, and xs_0 itself is neither *used* nor *updated*. Similarly, a in `elem` is *used* but not *updated*, and is therefore classified as *clean*. The list argument, ys_0 , in `sum` is also classified as *clean*, since its case-derived variables y and ys are *used* but ys_0 itself is neither *used* nor *updated*. Finally, the accumulator argument, x , in `sum` is classified as *tainted*, since it is both *used* and *updated*.

4.3.4 Classifying Operations

Once all the arguments of e are classified as *global*, *clean*, or *tainted*, we can proceed to classify the non-recursive application subexpressions of e , i.e. the *operations* in e . Those *operations* that take one or more variables that are classified as *tainted* are themselves classified as *obstructive*. All other *operations* are classified as *unobstructive*.

Definition 4.3.6 (Operation Obstructiveness). *Given some program p , some program environment Γ_p , and some fixpoint expression $e = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow e')$ in p , an operation $g(e_1, \dots, e_m) \ll e'$, when $g \neq f$, is classified as *obstructive* when $\exists i \in [1, m]$ such that $f(\vec{e}_f) \ll e_i \vee \exists y, y \ll e_i$ where y is classified as *tainted*. The operation is classified as *unobstructive* otherwise.*

For example, `sudoku` has only one operation, `(solve p)`; since p is classified as *clean*, `(solve p)` is classified as *unobstructive*. Similarly, the sole operation in `elem`, `(x == a)`, is classified as *unobstructive* since both x and a are classified as *clean*. Finally, the sole operation in `sum`, `(x+y)`, is classified as *obstructive* since x is classified as *tainted*. A similar result can be obtained when `sum` is rewritten to remove the accumulator:


```

1  sum ys0@[] = 0
2  sum ys0@(y:ys) = y + sum ys

```

Here, slicing with criterion ys_0 yields:

$$\Sigma^{\text{sum}}|ys_0 = \{y, ys\}$$

and is therefore classified as *clean*. However, the $(+)$ operation in Line 2 is classified as *obstructive*, since the second argument of $(+)$ has as a subexpression a recursive call.

4.4 Implementation

In this section, we give an overview of our implementation, highlighting differences from the description of our approach. Our implementation can be found at <https://adb23.host.cs.st-andrews.ac.uk/fhpc17-artefact.zip>.

Our prototype is implemented in Erlang, and comprises a parser and a classifier. The parser is implemented using the Leex and Yecc lexer and parser generators provided by the Erlang standard libraries. To avoid shift/reduce conflicts in the generated parser, an expression delimiter (`end`) is added to some expression types. The BNF grammar our prototype accepts is shown in Figure 4.5.

Once a file is parsed, our prototype proceeds to inspect each operation in each top-level fixpoint definition. For each fixpoint definition, it takes a slice of each variable declared in the fixpoint function. The classifier reflects the slicing algorithm in Section 4.3.2. It then classifies each variable according to the definition in Section 4.3.3. Finally, it classifies each operation according to the definition in Section 4.3.4 and the variable classifications derived in the previous stage. As a result, our prototype prints each operation and its classification.

4.5 Refactoring to Introduce Map Operations

The analysis presented in Section 4.3 classifies operations according to obstructiveness, but does not rewrite the fixpoint expression to introduce

```
 $\langle varlist \rangle := \text{var} \mid \text{var} \text{ ', ' } \langle varlist \rangle$   
 $\langle arglist \rangle := \langle expression \rangle \mid \langle expression \rangle \text{ ', ' } \langle arglist \rangle$   
 $\langle applist \rangle := \text{'(' } \langle arglist \rangle \text{' )'}$   
 $\langle listexp \rangle := \text{'nil' } \mid \text{'(' } \langle expression \rangle \text{ '::' } \langle expression \rangle \text{' )'}$   
 $\langle ifexp \rangle := \text{'if' } \langle expression \rangle \text{' then' } \langle expression \rangle \text{' else' } \langle expression \rangle \text{' end'}$   
 $\langle caseexp \rangle := \text{'case' var 'of' 'nil' '->' } \langle expression \rangle \text{ ', '}$   
 $\quad \text{var '::' var '->' } \langle expression \rangle \text{' end'}$   
 $\langle appexp \rangle := \text{var } \langle applist \rangle$   
 $\quad \mid \langle absexp \rangle \langle applist \rangle$   
 $\quad \mid \langle fixexp \rangle \langle applist \rangle$   
 $\langle absexp \rangle := \text{'\ ' ' ( ' } \langle varlist \rangle \text{ ' )' '->' } \langle expression \rangle \text{' end'}$   
 $\langle fixexp \rangle := \text{'fix' } \langle absexp \rangle$   
 $\langle expression \rangle := \text{'true'}$   
 $\quad \mid \text{'false'}$   
 $\quad \mid \text{int}$   
 $\quad \mid \text{var}$   
 $\quad \mid \langle listexp \rangle$   
 $\quad \mid \langle ifexp \rangle$   
 $\quad \mid \langle caseexp \rangle$   
 $\quad \mid \langle appexp \rangle$   
 $\quad \mid \langle absexp \rangle$   
 $\quad \mid \langle fixexp \rangle$   
 $\langle statement \rangle := \text{var ':='} \langle absexp \rangle$   
 $\quad \mid \text{var ':='} \langle fixexp \rangle$   
 $\langle program \rangle := \langle statement \rangle \mid \langle statement \rangle \text{' ; ' } \langle program \rangle$ 
```

Figure 4.5: Corresponding grammar for E accepted by the prototype Erlang implementation of our analysis.

a `map` operation. In this section, we describe a refactoring that lifts an *unobstructive* operation in a fixpoint expression into a `map` operation. While `map` has multiple possible implementations, according to the number and type of data structures being traversed, the refactoring defined in this section will use the standard `map`, defined over a single list.

```

1 map g []      = []
2 map g (x:xs) = g x : map g xs

```

Introducing a call to `map` places additional conditions on the operations we can lift into a `map` operation. We describe these conditions in Section 4.5.1.

The refactoring to lift operations and introduce a `map` operation is a composite refactoring, i.e. a multi-stage refactoring made up of multiple refactorings [81], with 4 stages. Given a fixpoint expression, f , that traverses a list, xs_0 , and an operation, g , that is a subexpression of f , the refactoring enacts the follow stages:

1. Duplicate the definition of f , perhaps using the *Duplicate Function* refactoring. The new, duplicated, definition, f' , becomes the helper function to f . f is rewritten, perhaps using the *Fold/Unfold Definition* refactoring, as a function expression whose body is a call to f' . For example, a given f ,

```

1 f xs0@[] = 0
2 f xs0@(x:xs) = g x + (f xs)

```

is refactored

```

1 f xs0 = f' xs0
2
3 f' xs0@[] = 0
4 f' xs0@(x:xs) = g x + (f' xs)

```

where f is duplicated to create the helper function, f' , and f becomes a call to f' .

2. The fixpoint function of the `map` operation that is to be introduced is constructed from g , the operation to be lifted. g is first lifted into

a lambda that takes one argument, i.e. the head of xs_0 , x . While g may take more than one argument, arguments other than x are guaranteed to be in scope as a consequence of the conditions in Section 4.5.1:

- Literals, i.e. booleans, integers, and nil_τ , are inherently in scope.
- Variables come in three possibilities: used but not updated; updated but not used; and updated and used. Variables that are used but not updated have the same value for all successive recursive calls to f' , so can be free variables in the constructed lambda, deriving their initial values in f . Variables that are updated but not used cannot occur as a subexpression to the lifted operation since this would contradict the definition of variable usage. Similarly, variables that are both used and updated cannot occur as a subexpression to the lifted operation since this would contradict the condition that g is considered to be *unobstructive*, and, by extension, contradict the definition of *clean* (Definition 4.3.5).
- Complex expressions, i.e. cons-, application-, case-, if-, lambda-, and fixpoint expressions, are in scope if their subexpressions are in scope.

For example, considering the above f' , the lambda constructed by lifting g is: $(\backslash x \rightarrow g\ x)$.

3. The map operation is now constructed by passing the constructed lambda and xs_0 to `map`. For example:

```
1 (map (\x → g x) xs0)
```

The constructed map operation is then passed as an additional argument, ys_0 , to f' in f , and the definition of f' is updated to accept an additional argument. Pattern matching on ys_0 mimics the pattern matching over xs_0 , and the tail of ys_0 is passed in each recursive call. For example, in

```

1  f xs0 = f' xs0 (map (\x -> g x) xs0)
2
3  f' xs0@[] ys0@[] = 0
4  f' xs0@(x:xs) ys0@(y:ys) = g x + (f' xs ys)

```

the `map` operation is passed as a section argument to `f'`, and `f'` has been updated to traverse the new list by pattern matching `nil` and `cons` constructors respectively, and the tail of `ys0`, `ys`, is passed as the second argument to the recursive call. Where case-expressions are used to discriminate `xs0`, as in E , a case-expression is introduced to `f'` within both branches of the case-expression over `xs0`. The original body of each case-split is moved into the matching branch of the case-expression over `ys0`, and the other branch is undefined. For example, given a rewritten `f'` to use case-expressions instead of pattern matching:

```

1  f' xs0 ys0 =
2    case xs0 of
3      [] -> case ys0 of
4              [] -> 0
5              (y:ys) -> undefined
6      (x:xs) -> case ys0 of
7                  [] -> undefined
8                  (y:ys) -> g x + (f' xs ys)

```

In Line 3 a case-expression over `ys0` is introduced where the original expression, (0) , is used in the `nil` branch. By definition the `cons`-branch of the `ys0` case-split cannot be reached, so it is left undefined in Line 4. Similarly, in Line 5, a case-expression over `ys0` is introduced with the original expression, $(g\ x + (f'\ xs))$, is used in the `cons`-branch of the `ys0` case-split, and the unreachable `nil` branch is left undefined. The recursive call in the `cons`-branch is updated to pass `ys` as a second argument.

4. Finally, substitute (or *fold*, in the transformational sense [27]) `g` for the (equivalent) head of `ys0`. For example, in

```

1  f' xs0@[] ys0@[] = 0
2  f' xs0@(x:xs) ys0@(y:ys) = y + (f' xs ys)

```

(g x) in Line 2 has been substituted for y.

The refactoring can be applied for multiple operations in f by constructing a `map` operation for each operation to be lifted. The refactoring should take the outermost *unobstructive* operation; i.e. there should be no h such that h is an *unobstructive* operation and g is a subexpression of h .

4.5.1 Conditions

The conditions for lifting an operation into a map operation, using the definition of `map` above, is as follows:

- The operation to be lifted is classified as *unobstructive*, and all subexpressions that are operations are also classified as *unobstructive*.
- The operation must have exactly one argument with a subexpression that is a variable y , such that there exists a case-expression in f :

$$\text{case } ys_0 \text{ of } \text{nil}_\tau \rightarrow e_{\text{nil}}, \text{cons}_\tau y \, ys \rightarrow e_{\text{cons}}$$

- f must traverse exactly one list argument, xs_0 , and f must contain exactly one case-expression discriminating on xs_0 . The tail of the list, ys , must be passed to all recursive calls as the i^{th} argument, where xs_0 is declared as the i^{th} argument in f .

The requirement that only one list be traversed for the refactoring to apply may be relaxed if, e.g., a `map` over two lists is to be introduced. We conjecture that alternative `map` definitions will require specific conditions, according to their definition. For example, the `map`:

```

1  map g xs0@[] = []
2  map g xs0@(w:x:xs) = g w x : map g xs

```

would require that exactly two case-expressions over xs_0 occur in f . Alternatively, the `map`:

```

1 map g xs₀@[] ys₀@[] = []
2 map g xs₀@(x:xs) ys₀@(y:ys) = g x y : map g xs ys

```

more commonly known as `zipWith`, would require exactly two case-expressions over two list arguments in f .

4.5.2 Additional Cleaning Stages

By lifting *unobstructive* operations into `map` operations, some function arguments may no longer be necessary in the definition of f' . Additional refactorings can be used to remove those unnecessary arguments, or at the most extreme, remove an unnecessary helper function.

Once f is refactored, and all *unobstructive* operations are lifted into maps and passed to f' , those arguments to f' which are neither *used* nor *updated* in the definition of f' may be eliminated, perhaps by the *Eliminate Function Argument* refactoring. In the case of list arguments, we extend this to mean neither the head nor the list itself is used, and the tail is only used in all recursive calls as a means of traversing the list. For example, in

```

1 f xs₀ = f' xs₀ (map (\x -> g x) xs₀)
2
3 f' xs₀@[] ys₀@[] = 0
4 f' xs₀@(x:xs) ys₀@(y:ys) = g x + (f' xs ys)

```

we observe that x and $xs₀$ are not *used* in f' , nor are they *updated*. xs *occurs* in the recursive call, but only as a means to traverse $xs₀$. f' may therefore be refactored to eliminate $xs₀$:

```

1 f' ys₀@[] = 0
2 f' ys₀@(y:ys) = y + (f' ys)

```

Where f' contains no other (*obstructive*) operations as subexpressions, f' itself may be eliminated, where the call to f' in f' is replaced with the call to the introduced `map` operation. Recall the definition of `sudoku`,

```

1 sudoku ps₀@[] = []
2 sudoku ps₀@(p:ps) = solve p : sudoku ps

```

where the operation `(solve p)` may be lifted into a map:

```
1 sudoku ps0 = sudoku' ps0 (map (\p -> solve p) ps0)
2
3 sudoku' ps0@[] qs0@[] = []
4 sudoku' ps0@(p:ps) qs0@(q:qs) = q : sudoku' ps qs
```

Here, no other operations occur within `sudoku'`, meaning the call to `sudoku'` in `sudoku` may be wholly replaced by the introduced map operation.

```
1 sudoku ps0 = map (\p -> solve p) ps0
```

Another simplification can be applied when multiple disjoint map operations are introduced. As a fusion step [48], it is possible to combine those map operations that act over the same list, perhaps using the standard *Tupling* transformation. For example, given a refactored `f` with two disjoint *unobstructive* operations, `g` and `h`,

```
1 f xs0@ =
2   f' (map (\x -> g x) xs0@) (map (\x -> h x) xs0)
3
4 f' ys0@[] zs0@[] = 0
5 f' ys0@(y:ys) zs0@(z:zs) = y + (z + (f ys zs))
```

the map operations over `ys0` and `zs0` may be *tupled*, or *zipped*:

```
1 f xs0@ = f' (map (\x -> ((g x), (h x))) xs0)
2
3 f' ws0@[] = 0
4 f' ws0@((y,z):ws) = y + (z + (f ys zs))
```

When using the standard definition of `map`, we note that only one list argument may be traversed as a condition of the refactoring. This means that *all* introduced map operations will, by definition, apply to the same list, and so *all* map operations may be tupled. This may not hold for alternative map definitions.

4.6 Examples

In this section, we demonstrate our approach on 13 examples, including a suite of five parallel benchmarks and nine other examples. We describe our experimental setup for our parallel benchmarks in Section 4.6.1. In Section 4.6.2, we describe our five parallel benchmarks, including reporting the parallel speedups we achieve, and reporting the results of our analysis. The examples in described in Section 4.6.3 may not benefit from parallelisation, but are used to illustrate our approach. All examples have been translated to our expression language, *E*, by hand, and then passed to our prototype implementation described in Section 4.4.

4.6.1 Experimental Setup

Our speedup results are given as an average of five runs on *corryvreckan*, a 2.6GHz Intel Xeon E5-2690 v4 machine with 28 physical cores and 256GB of RAM. This machine allows *turbo boost* up to 3.6GHz, and supports automatic dynamic frequency scaling between 1.2–3.6GHz. Our parallel benchmarks have been compiled using GHC 7.6.3 on Scientific Linux using the flags: `-rtsopts`, `-threaded`, and `-feager-blackholing`, and also use `-O2` for `sudoku` and `-O` for all other examples. We found that these settings gave the best general parallel performance. Our measurements represent the average of five runs and are taken from the `-s` Haskell RTS option, which records execution statistics. Timing information is taken from two sources: *mutator time* (`MUT`, indicating time spent purely on executing the program) and *total time* (`Total`, elapsed real-time while running the program), both in seconds. Where multiple parallelisations are possible, we execute the one that reports best parallel performance. We use a separate, standard desktop machine, *neptune*, to test our prototype implementation. *neptune* is a 2.7GHz Intel Core i5 machine with 8GB of RAM, running Mac OS X 10.11.6 and Erlang 19.2.3.

4.6.2 Parallel Benchmarks

We demonstrate our approach using five parallel benchmarks: `sudoku` from [86]; and `sumeuler`, `queens`, `nbody`, and `matmult` from the

n	MUT	SD	Total	SD	Sparks	Heap	Residency
16	17.52	0.07	18.71	0.10	0	3.39	0.214
160	176.35	2.44	187.75	2.45	0	33.95	1.788
320	348.39	0.42	371.15	0.41	0	67.89	3.826
480	521.63	1.15	555.65	1.13	0	101.84	5.664
640	699.49	2.15	744.99	2.20	0	135.78	7.814
800	878.83	1.95	935.25	2.15	0	169.73	8.822

Figure 4.6: Sequential mutator (MUT) and total (Total) times in seconds (with standard deviations), total number of sparks for parallel version, heap allocation in gigabytes, and residency in bytes for *sudoku* on *corryvreckan*. n refers to the number of puzzles, and is a multiple of 10^3 .

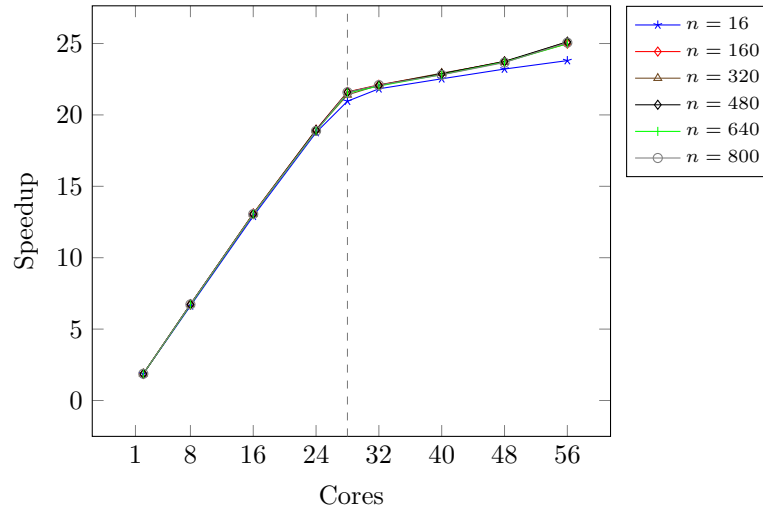


Figure 4.7: *sudoku*, speedups on *corryvreckan*, a 28-core hyperthreaded Intel Xeon server, dashed line shows extent of physical cores using reported mutator (MUT) time. n denotes the number of *sudoku* puzzles solved, and is a multiple of 10^3 .

NoFib suite [99]. We have translated the Haskell definitions to our simple expression language defined in Section 4.2, unfolded the parallelised `map` operations, removing any parallel constructs, and finally, applied our prototype analyser to the resulting function definitions.

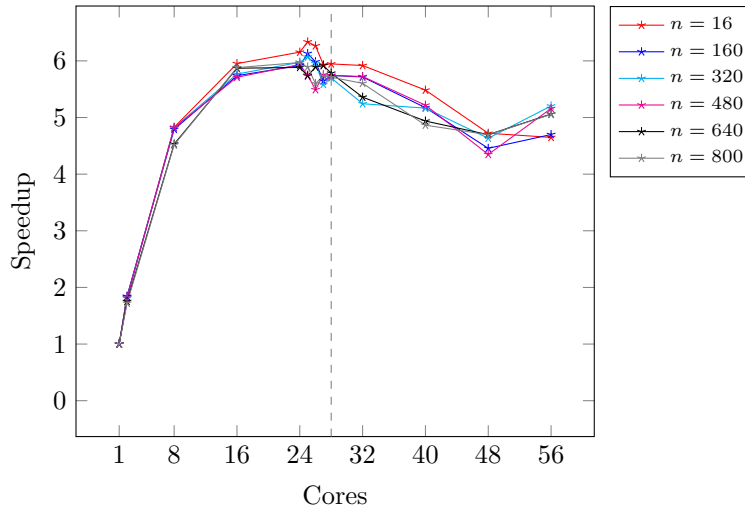


Figure 4.8: sudoku, speedups on *corryvreckan* using reported total (Total) time, dashed line shows extent of physical cores. n is a multiple of 10^3 .

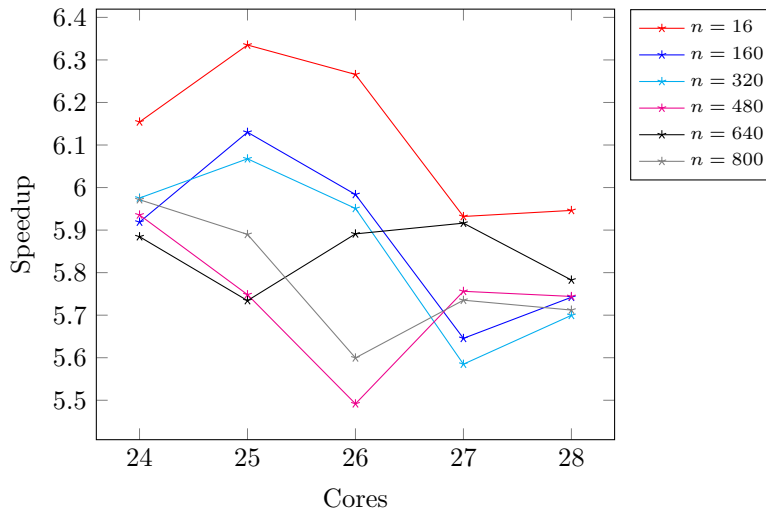


Figure 4.9: sudoku, speedups on *corryvreckan* using reported total (Total) time. Zoom of Figure 4.8. n is a multiple of 10^3 .

Sudoku

Recall that `sudoku` solves a list of Sudoku puzzles, is defined:

```
1 sudoku [] = []
2 sudoku (p:ps) = solve p : sudoku ps
```

In [86], `sudoku` consists of nine Haskell files that comprise a sequential

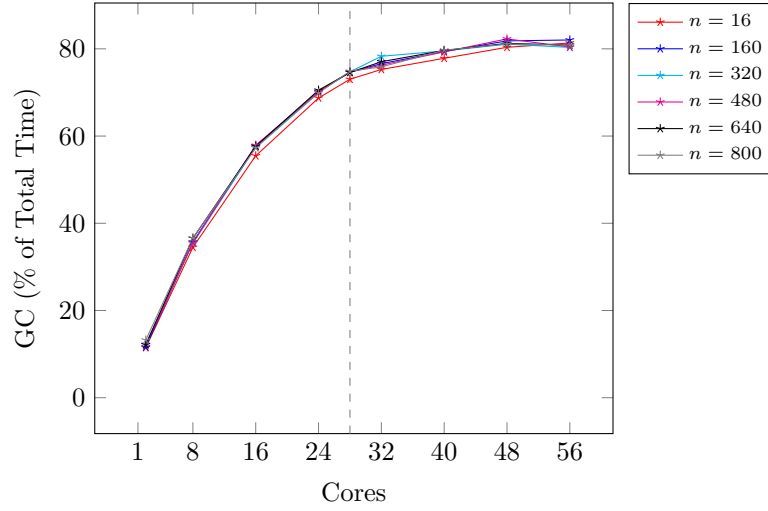


Figure 4.10: *sudoku*, percentage of time spent on garbage collection on *corryvreckan* using reported total (Total) time, dashed line shows extent of physical cores. n is a multiple of 10^3 .

implementation and various parallelisations. The sequential implementation comprises the module `Sudoku` (150 lines) that contains functions to solve individual puzzles, and the file `sudoku1.hs` (16 lines) that contains one function that reads in a list of puzzles and solves them by applying `solve`. `sudoku1.hs` contains a single function: `main`, that contains an explicit `map` operation to apply `solve` to each `Sudoku` problem. We have derived the above definition of `sudoku` by lifting this `map` operation into a function, and inlining the `map`. `Sudoku` contains 10 functions with 10 explicit `map` operations, four explicit `fold` operations, three explicit `zip` operations, two explicit `filter` operations, two implicit `filter` operations, and one implicit `map` operation. We include list comprehensions, but do not count specialisations of `map` or `fold` operations, e.g. `length` or `maximum`. The parallel versions comprise four versions using the `Par Monad`, and four using the `Eval Monad` and the `Strategies` library. All parallel versions contain focus on the parallelisation of `sudoku`.

We recall that `sudoku` has the slice, $\Sigma^{\text{sudoku}|xs_0} = \{x, xs\}$. The reachable arguments `p` and `ps` are used in the first and second arguments to the `cons`, with `ps` passed as the first and only argument to the recursive call. `ps0` is not considered to be updated since `ps` is the tail of `ps0`. `ps0` is classified as *clean*. `sudoku` has only one operation: `(solve p)`. Since `p`

is classified as *clean*, `sudoku` is therefore classified as *unobstructive*. Since `(solve p)` is the only operation in `sudoku` we can rewrite `sudoku` as a `map` and then parallelise it, for example, by using the `Par Monad`,

```
1 sudoku ps = runPar $ parMap solve ps
```

as shown in Section 4.1.

We executed the parallel version of `sudoku` for $n = 16,000$ and between 160,000 and 800,000 at intervals of 160,000. Figure 4.6 gives average sequential times (with standard deviations), the number of sparks, heap allocation, and residency for all n . Unlike the other parallel benchmarks, `sudoku` does not report any sparks for parallel versions since we use the `Par Monad`. Figures 4.7 and 4.8 give the corresponding speedups using mutator and total time. We achieved maximum speedups of $25.13\times$ when $n = 320,000$ on 56 hyperthreaded cores (`MUT`), and $6.33\times$ when $n = 320,000$ on 26 cores (`Total`). Figure 4.7 shows that varying n has little effect on speedup, but that increasing the number of cores produces good speedups and hence that the example shows good scalability. Our results show that some benefit can be achieved on hyperthreaded cores, but that this is insignificant when compared with increasing the number of physical cores. The total times show that speedups are only half those that would be expected from the mutator times alone. This is likely due to significant garbage collection overheads. Figure 4.10 shows that the percentage of the total time spent on garbage collection increases with the number of cores. The increased garbage collection time is likely due to repeated generation of lists in `solve`. Unlike mutator time speedups, the total time speedups plateau before 28 cores. Figure 4.9 shows that there is some variation in speedups between 24 and 28 physical cores. This reflects an underlying variation in execution times, e.g. for $n = 480,000$, there is a standard deviation (σ) of 12.31s for 25 cores, $\sigma = 17.78s$ for 26 cores, and $\sigma = 2.25s$ for 27 cores. This is likely to be due to inconsistent scheduling as the machine becomes saturated. With hyperthreading enabled, speedups reduce as more cores are added due to increased garbage collection times. On 48 cores there is a notable drop in speedups, again likely due to poor scheduling.

n	MUT	SD	Total	SD	Sparks	Heap	Residency
1	0.06	0.02	0.06	0.02	3	67795856	68256
5	1.29	0.01	1.31	0.01	11	1958285432	214792
10	5.34	0.03	5.39	0.03	21	8296937744	224584
15	12.43	0.03	12.54	0.03	31	19280868336	498624
20	22.82	0.08	23.03	0.07	41	35050884184	761720
25	36.29	0.03	36.6	0.04	51	55705003000	1024160
30	53.24	0.18	53.7	0.18	61	81319554136	1228304
35	73.36	0.3	73.99	0.3	71	111956760792	1428304
40	96.77	0.04	97.6	0.04	81	147666965368	1612232
45	123.98	0.13	125.04	0.13	91	188497463480	1612232
50	154.33	0.25	155.65	0.26	101	234487830776	1778184

Figure 4.11: Sequential mutator (MUT) and total (Total) times in seconds (with standard deviations), total number of sparks for parallel version, and heap allocation and residency in bytes for `sumeuler` on *corryvreckan*. n represents the bounding value of the input list $[1, n]$, and is a multiple of 10^3 .

Sumeuler

`sumeuler` calculates Euler’s totient function for a list of integers and sums the results.

```

1  sumeuler xs0@[] = 0
2  sumeuler xs0@(x:xs) = euler x + (sumeuler xs)

```

In the NoFib suite, `sumeuler` consists of three Haskell files: `ListAux` (41 lines), `SumEulerPrimes` (36 lines), and `SumEuler` (290 lines). Collectively, these introduce 31 functions, 12 of which have explicit *maps* or *folds* as part of their definitions.

As with `sudoku`, we first slice `sumeuler` for its only argument, xs_0 , producing the slice: $\Sigma^{\text{sumeuler}}|_{xs_0} = \{x, xs\}$. As before, xs_0 is classified to be *clean* since both the case-split variables of xs_0 are used in the body of `sumeuler`, and xs_0 itself is not updated. Two operations exist as subexpressions to `sumeuler`: `(euler x)` and the application of `(+)` in Line 2. `(euler x)` takes a single *clean* argument, i.e. x , and is classified as *unobstructive*. Conversely, `(+)` takes two arguments, where the second argument comprises a recursive call, and is classified as

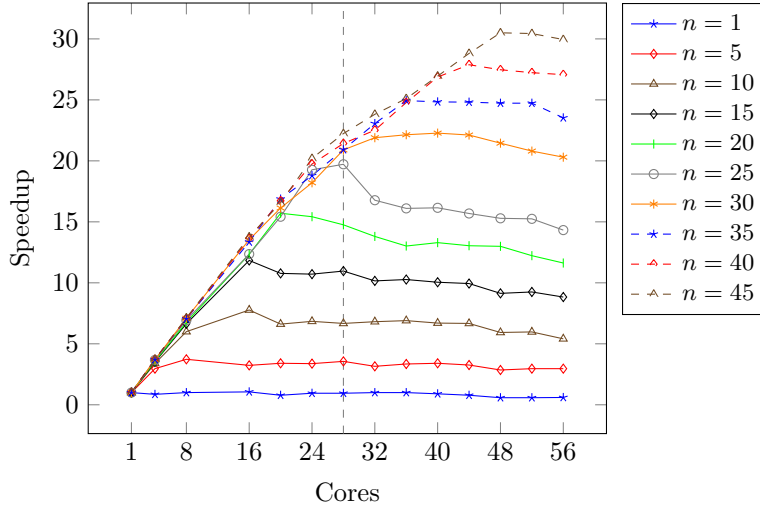


Figure 4.12: Speedups for `sumeuler` on `corryvreckan` using reported mutator times. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores. n is a multiple of 10^3 .

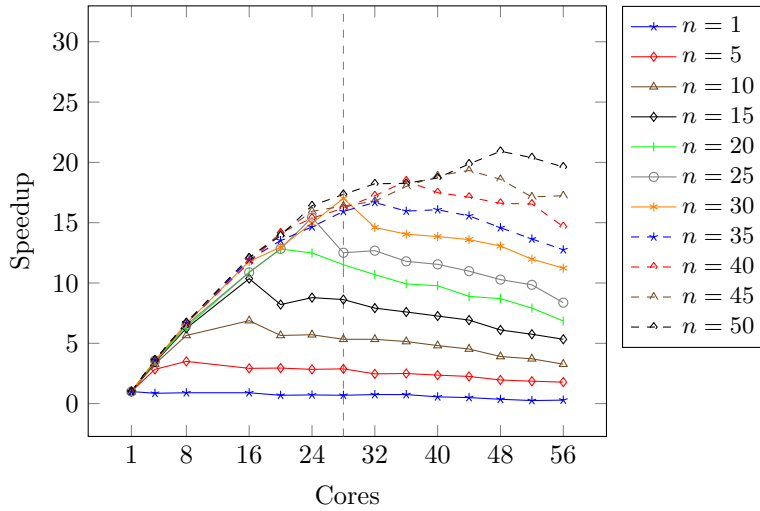


Figure 4.13: Speedups for `sumeuler` on `corryvreckan` using reported total (Total) times. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores. n is a multiple of 10^3 .

obstructive. (`euler x`) can be lifted into a `map` operation. Where, with `sudoku` we spawned a thread for each element in the input list, here we introduce *chunking*. Chunking groups elements into sublists. Evaluation of each sublist is then performed in parallel. Chunking can help to ensure that there is enough work per thread, so increasing *granularity*. We are also able to take advantage of the associativity of `(+)`, summing each

chunk before summing the result of all chunks. We can then parallelise `sumeuler`.

```
1 sumeuler :: [[Int]] -> Int
2 sumeuler xs =
3   sum (map (sum . map euler) xs
4         `using` parList rdeepseq)
```

We executed `sumeuler` for $n = 1,000$ and between 5,000 and 50,000 at intervals of 5,000. We chose a chunk size of 500 since it produced the best speedup. Figure 4.11 gives average sequential times (with standard deviations), the number of sparks, heap allocation, and residency for all n . All sparks are converted for all n . Figures 4.12 and 4.13 give speedups for `sumeuler` using mutator and total time. We achieve maximum speedups of 30.50 for $n = 50,000$ on 48 virtual cores (mutator) and 20.92 for $n = 50,000$ on 48 virtual cores (total). Our results show that `sumeuler` scales with both the number of cores and with the size of n . Unlike `sudoku`, mutator time speedups generally plateau due to lack of work and garbage collection has a minimal effect on `sumeuler` speedups. The most notable effect occurs for $n = 15,000$ and $n = 20,000$ on 20 cores, where speedups reduce noticeably. These results suggest that `sumeuler` requires relatively large n if parallelism is to be worthwhile. Sequentially, when $n = 50,000$ `sumeuler` has an average runtime of 154.33s ($\sigma = 0.25s$) and 155.65s ($\sigma = 0.26s$) for mutator and total time, respectively.

N-Queens

The `queens` problem asks how n queens can be placed on a chess board of size n^2 such that no queen may take another according to the usual rules of chess.

```
1 queens nq = gen 0 []
2
3 gen nq n b
4   n >= nq = [b]
5   otherwise = genloop nq n (gennext [b])
6
```


n	MUT	SD	Total	SD	Sparks	Heap	Residency
11	0.1	0.03	0.1	0.03	101	95722624	64088
12	0.42	0.08	0.43	0.08	122	569715376	1693112
13	2.13	0.02	2.19	0.03	145	3590068664	9854376
14	13.65	0	14.1	0.01	170	24108663032	59041312
15	96.36	0.17	103.11	0.18	197	171560553832	479706512
16	717.66	0.81	821.03	0.35	226	1293635766936	3097146864

Figure 4.14: Sequential mutator (MUT) and total (Total) times in seconds (with standard deviations), total number of sparks for parallel version, and heap allocation and residency in bytes, for *queens* on *corryvreckan*. n represents the number of queens to be placed on a $n \times n$ chessboard.

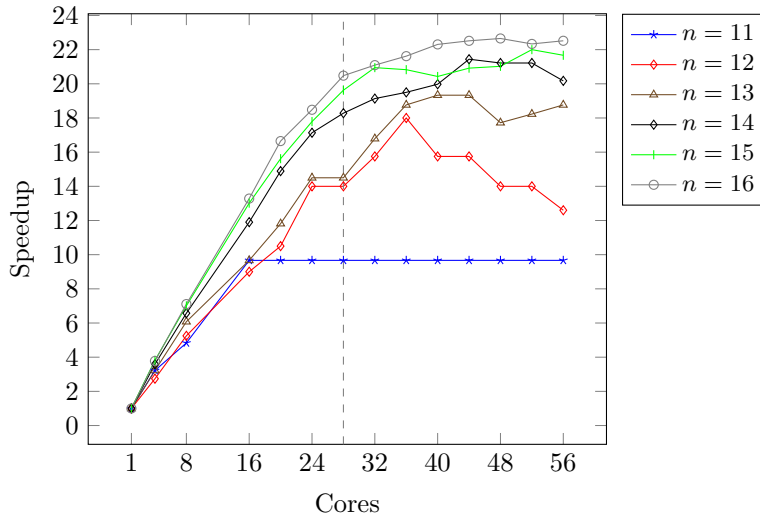


Figure 4.15: Speedups for *queens* on *corryvreckan* using reported mutator (MUT) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

```

7  genloop nq n bs0@[] = []
8  genloop nq n bs0@(b:bs) = gen nq (n+1) b ++ genloop bs

```

Here, `nq` is the number of queens, and `gennext` calculates the next safe position on the board to place a queen. In the *NoFib* suite, *queens* consists of one Haskell file, *Main*, that is 42 lines long and defines 6 functions. Two of these 6 functions have explicit `map` and `fold` operations (including list comprehensions), and an explicit call to `iterate`. One other function, *safe*, is an *implicit* `foldr` over lists.

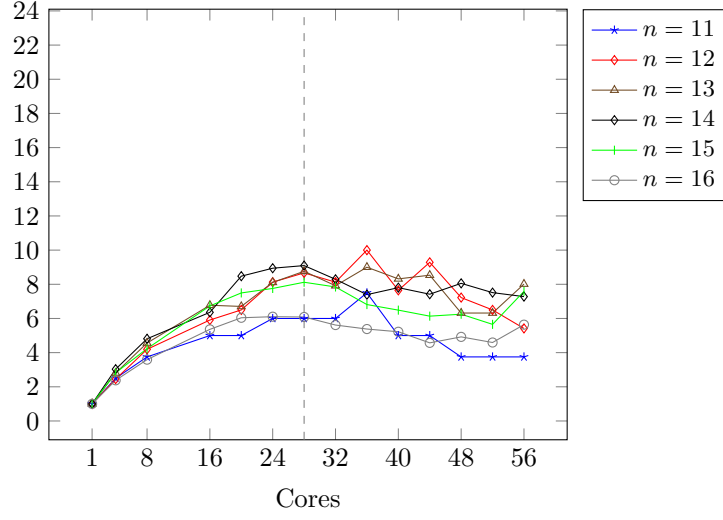


Figure 4.16: Speedups for `queens` on *corryvreckan* using reported total (Total) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

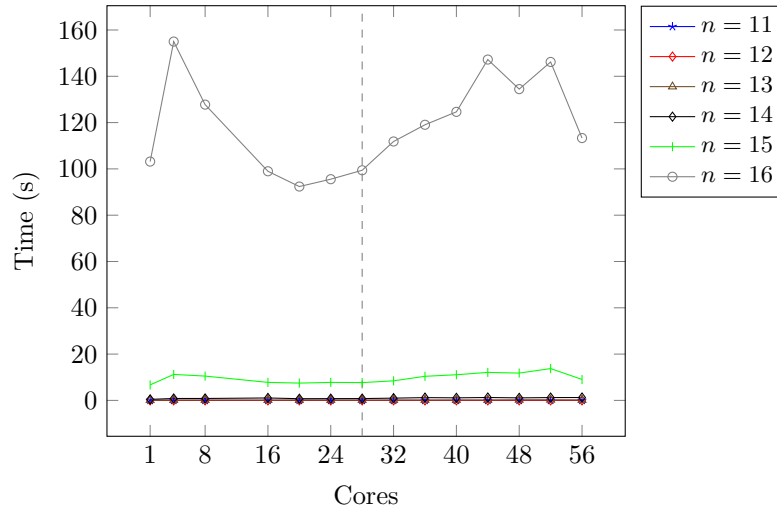


Figure 4.17: Garbage collection times for `queens` on *corryvreckan*. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores.

Since neither `queens` nor `gen` are recursive functions, we instead slice for all the arguments of `genloop`.

$$\Sigma^{\text{genloop}|nq} = \{nq\} \quad \Sigma^{\text{genloop}|n} = \{n\} \quad \Sigma^{\text{genloop}|bs_0} = \{b, bs\}$$

All three arguments, `nq`, `n`, and `bs0`, are classified as *clean* since both `nq` and `n` are *used* but not *updated*; and both `b` and `bs` are *used* but `bs0` is

neither *used* nor *updated*. `genloop` has three subexpressions: *i*) $(n+1)$ in the call to `gen`; *ii*) $(\text{gen } nq \ (n+1) \ b)$ in the first argument to $(++)$; and *iii*) the top-level $(++)$. Both $(n+1)$ and $(\text{gen } nq \ (n+1) \ b)$ are classified as *unobstructive* since only *clean* variables occur in the expressions passed to $(+)$ and `gen` respectively. We additionally note that $(\text{gen } nq \ (n+1) \ b)$ contains the *unobstructive* operation $(n+1)$ as a subexpression. Hypothetically, if $(n+1)$ had been classified as *obstructive*, then $(\text{gen } nq \ (n+1) \ b)$ must also be classified as *obstructive*. While no *tainted* variables occur in the arguments to the append operation, its second argument is a recursive call, which results in an *obstructive* classification. Although it is possible to lift either *unobstructive* operations into a map operation, we choose to lift the operation which does not occur as a subexpression of any *unobstructive* operation, i.e. $(\text{gen } nq \ (n+1) \ b)$, in order to maximise the amount of work that is done in parallel.

```

1  genloop nq n bs0 =
2    genloop' nq n (map (gen nq (n+1)) bs0)
3
4  genloop' cs0@[] = []
5  genloop' cs0@(c:cs) = c ++ genloop' cs

```

Similarly to `f` and `multMatricesTr` in `matmult`, we observe that `gen` and `genloop` could be merged into a single definition.

```

1  gen 0 b xs0@[] = [b]
2  gen n b xs0@[] = []
3  gen n b xs0@(x:xs) =
4    (gen (n-1) x (gennext [x])) ++ (gen n b xs)

```

This definition eliminates the mutual recursion of `gen` and `genloop`, where `xs0` traverses two different lists and `n` acts as an additional bound on the recursion. As before, our technique will (correctly) classify all operations as *obstructive*. In this example, it is the recursive call in the first argument to $(++)$ that introduces an update to all variables.

Our translation of the `queens NoFib` implementation comprises two recursive functions with 11 operations, and 2 recursive calls. Despite the high number of operations, there are only two *unobstructive* operations

in the translation. The majority of operations are found in the definition of `safe`.

```

1  safe x d ys0@[]      = True
2  safe x d ys0@(q:l)   =
3    x /= q && x /= q+d && x /= q-d && safe x (d+1) l

```

Slicing `safe` for all arguments,

$$\Sigma^{\text{safe}|x} = \{x\} \quad \Sigma^{\text{safe}|d} = \{d, \bar{d}\} \quad \Sigma^{\text{safe}|ys_0} = \{q, l\}$$

we classify `x` and `ys0` as *clean* and `d` as *tainted*. Here, the *tainted* classification of `d` results in *obstructive* classifications of all but one infix inequality operations. Hypothetically, if `d` had been classified as *clean*, we observe that how the infix (`&&`) operations are compiled can affect the classifications of operations in `safe`. Parsing (`&&`) as either left- or right-associative can produce different numbers of *obstructive* operations. Here, parsing (`&&`) as left-associative *minimises* the number of *obstructive* operations; i.e. when,

```

1  (x /= q && x /= q+d) && (x /= q-d && safe x (d+1) l)

```

only the topmost (`&&`) and its second argument are classified as *obstructive*. Conversely, parsing (`&&`) as right-associative *maximises* the number of *obstructive* operations; i.e. when,

```

1  x /= q && (x /= q+d && (x /= q-d && safe x (d+1) l))

```

all (`&&`) operations are classified as *obstructive*.

We executed `queens` for n ranging from 11 to 16, with a threshold depth of 2. Figure 4.14 gives average sequential times (with standard deviations), the number of sparks, heap allocation, and residency for all n . Figures 4.15 and 4.16 show speedups for `queens` in terms of mutator and total time. We achieve maximum speedups of 22.65 for $n = 16$ on 48 hyperthreaded cores and of 10.00 for $n = 12$ on 36 hyperthreaded cores. Sequentially, when $n = 16$, `queens` takes an average of 717.66s ($\sigma = 0.25s$) and 821.03s ($\sigma = 0.26$) for mutator and total time, respectively. When $n = 11$, the mutator time plateaus before the physical cores are exhausted,

likely due to a lack of work, as we also observed for `suneuler`. When $14 \leq n \leq 16$, `queens` scales well until 28 cores. When hyperthreading is enabled, we see some further improvement in speedup, albeit at a reduced rate when compared with those we obtain using physical cores. Interestingly, for $n = 13$ and $n = 14$, we see good scalability continue between 28 and 40 hyperthreaded cores. This may be due to the lack of chunking of `bs0`, resulting in inefficient use of cores up to 28 cores, so enabling full use of the machine. As with `sudoku`, total time speedups reveal an overall halving of performance gains, as a consequence of increased garbage collection time. Again, this is likely due to repeated generation of lists.

It is interesting to note that the definitions of `gen` and `genloop` may be merged as part of the unfolding transformation.

```

1  gen 0 b xs0@[] = [b]
2  gen n b xs0@[] = []
3  gen n b xs0@(x:xs) =
4    (gen (n-1) x (gennext [x])) ++ (gen n b xs)

```

This eliminates the mutual recursion, but also changes the classification of `n` and `b` to be *tainted*. Given the slices,

$$\Sigma^{\text{gen}|n} = \{n, \bar{n}\} \quad \Sigma^{\text{gen}|b} = \{b, \bar{b}\} \quad \Sigma^{\text{gen}|xs_0} = \{x, xs, \bar{xs}_0\}$$

all three arguments are both *used* and *updated* between recursive calls. The recursive call in the first argument of `(++)` on Line 3 introduces an update for all arguments between recursive calls. This results in all operations being classified as *obstructive*. This is correct because the same `xs0` is not traversed for all `n`. `gen` is a combination of two patterns: *iteration* and *map*. However, we are unable to produce the best parallelisation in this case: our approach detects only *map* operations.

N-Body

The `nbody` problem is to calculate the movement of bodies in m -dimensional space according to forces between them. The classical representation of this problem is between celestial bodies, modelling the effects of gravity

n	MUT	SD	Total	SD	Sparks	Heap	Residency
10	2.06	0.06	2.07	0.06	40	6310152	1197488
25	11.61	0.1	11.63	0.11	40	15670360	5216880
50	46.93	0.13	46.98	0.13	40	31270520	12311984
75	104.9	0.35	104.96	0.34	40	46870848	12547600
100	187.47	0.1	187.57	0.11	40	62471232	25082544
250	1164.34	0.68	1164.52	0.65	40	156226496	40215360

Figure 4.18: Sequential mutator (MUT) and total (Total) times in seconds (with standard deviations), total number of sparks for parallel version, and heap allocation and residency in bytes, for *nbody* on *corryvreckan*. n denotes the number of points to cluster, and is a multiple of 10^3 .

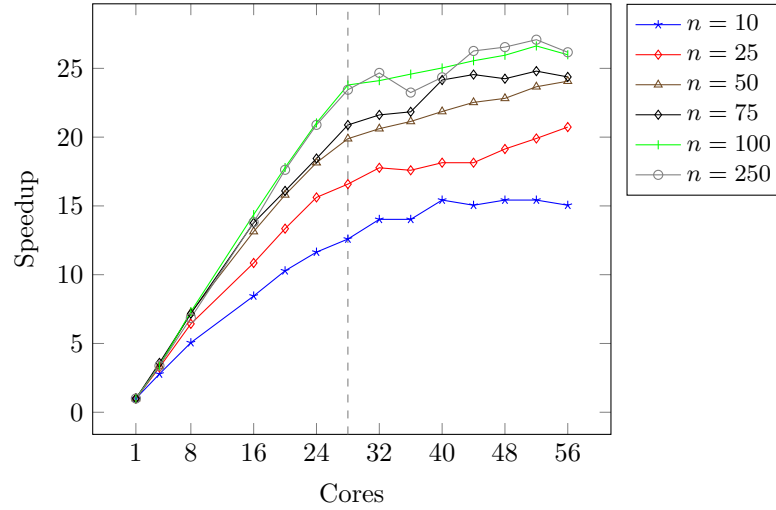


Figure 4.19: Speedups for *nbody* on *corryvreckan* using reported mutator (MUT) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores. n is a multiple of 10^3 .

on their trajectories. In the NoFib suite, *nbody* consists of two Haskell files: `Future` (16 lines) and `nbody` (135 lines) which collectively define 8 functions. Two of these 8 functions have an explicit `map` or `fold`.

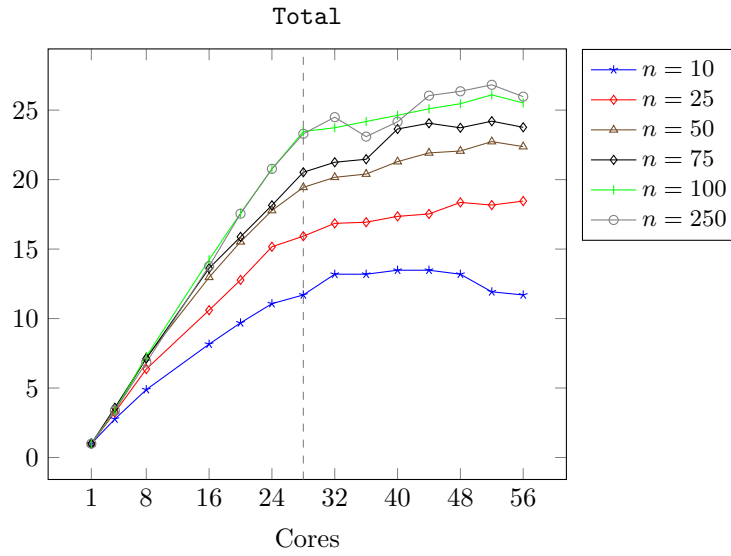


Figure 4.20: Speedups for `nbody` on *corryvreckan* using reported total (Total) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores. n is a multiple of 10^3 .

```

1  run n =
2    let initVecs = ...
3      chunk = ...
4    in fs n chunk initVecs [1,1+chunk..n]
5
6  fs n chunk initVecs ts0@[] = []
7  fs n chunk initVecs ts0@(t:ts) =
8    f initVecs [t..(min (t+chunk-1) n)]
9    ++ fs n chunk initVecs ts
10
11 f initVecs ts0@[] = []
12 f initVecs ts0@(t:ts) =
13   compute initVecs t : f initVecs ts

```

where `n` bodies are stored in an `Array` (which we convert internally to *cons*-lists), split into chunks, and their updated locations are calculated using `compute` in Line 11. Both `fs` and `f` are fixpoint expressions, which

we slice. Slicing for all four arguments of `fs` produces:

$$\begin{array}{ll} \Sigma^{fs|n} &= \{n\} & \Sigma^{fs|chunk} &= \{chunk\} \\ \Sigma^{fs|initVecs} &= \{initVecs\} & \Sigma^{fs|ts_0} &= \{t, ts\} \end{array}$$

All four arguments are *used* but not *updated*, and they are therefore classified to be *clean*. When the enumeration syntax is expanded to an application of `enumFromTo`, `fs` has six operations:

1. `(...++...);`
2. `(f initVecs [t..(min (t+chunk-1) n)]);`
3. `(enumFromTo t (min (t+chunk-1) n));`
4. `(min (t+chunk-1) n);`
5. `((t+chunk)-1);` and
6. `(t+chunk).`

`(...++...)` is classified as *obstructive* since its second argument comprises a recursive call; all other operations are classified as *unobstructive* since they take only *clean* variables and *unobstructive* operations as subexpressions. Slicing for all arguments of `f`,

$$\Sigma^{f|initVecs} = \{initVecs\} \quad \Sigma^{f|ts_0} = \{t, ts\}$$

we classify all arguments as *clean* since all of them are *used* but not *updated*. The sole operation, `(compute initVecs t)`, on Line 12 is therefore classified as *unobstructive*. `f` is rewritten as a `map` operation that is then parallelised.

We executed `nbody` for $n = 10,000$ and between 25,000 and 100,000 at intervals of 25,000. Figures 4.19 and 4.20 give speedups for `nbody` using `mutator` and total time, achieving maximum speedups of 27.08 for $n = 250,000$ on 52 hyperthreaded cores and 26.82 for $n = 250,000$ on 52 hyper-threaded cores, respectively. Sequentially, where $n = 250,000$ `nbody` takes an average of 1164.34s ($\sigma = 0.68s$) and 1164.52s ($\sigma = 0.65s$) for `mutator` and total time, respectively. Our results show good scaling for both n and the number of cores, with speedups increasing linearly

n	MUT	SD	Total	SD	Sparks	Heap	Residency
10	13.15	1.59	13.37	1.58	18	317.36	44416
15	56.09	5.12	56.33	5.14	27.5	713.75	44416
20	131.72	14.07	131.96	14.07	38	1268.62	44416
25	226.58	7.49	226.83	7.51	49.5	1835.83	44416
30	402.13	2.71	402.36	2.69	62	1801.41	44416
35	633.18	9.43	633.44	9.43	67	2451.61	44416
40	946.42	12.1	946.67	12.08	80.5	3201.84	44416
45	1344.39	16.09	1344.59	16.08	85.5	4052.05	44416
50	1823.05	0.96	1823.31	0.96	100	5002.25	44416

Figure 4.21: Sequential mutator (MUT) and total (Total) times in seconds (with standard deviations), total number of sparks for parallel version, heap allocation in megabytes, and residency in bytes, for `matmult` on *corryvreckan*. n denotes the size of the input matrix, $n \times n$, and is a multiple of 10^2 .

with n . As in other examples, performance gains do not grow as quickly with hyperthreading enabled. There is little difference between mutator and total reported speedups for `nbody`, showing that garbage collection is not significant for this example.

Matrix Multiplication

The NoFib benchmark uses the following implementation of matrix multiplication.

```
1 matmult m1 m2 = multMatricesTr m1 (transpose m2))
2
3 multMatricesTr [] m2 = []
4 multMatricesTr (r:rs) m2 =
5   f m2 r : multMatricesTr rs m2
6
7 f cs0@[] row = []
8 f cs0@(c:cs) row = prodEscalar2 row c : f cs row
```

Here, `transpose` transposes a matrix and `prodEscalar2` calculates the dot product of two lists. Since `matmult` is not a fix expression, we

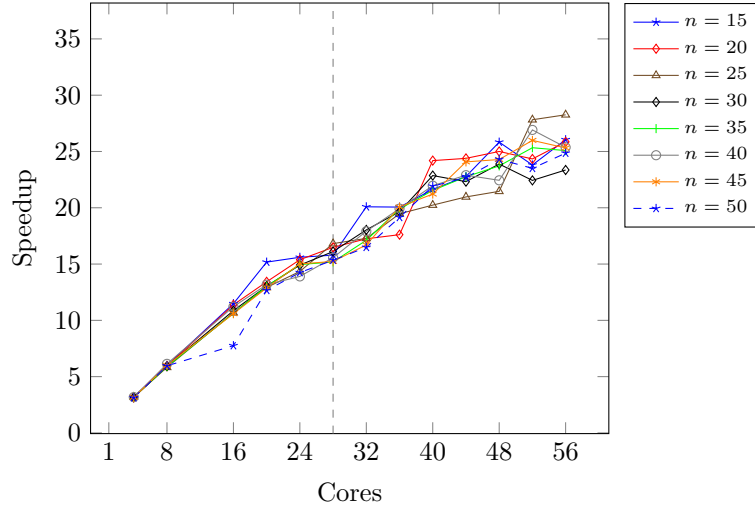


Figure 4.22: Speedups for `matmult` on `corryvreckan` using the row-wise parallelisation option and reported mutator (MUT) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores. n is a multiple of 10^2 .

slice for all the arguments to `multMatricesTr`.

$$\Sigma^{\text{multMatricesTr}|rs_0} = \{r, rs\} \quad \Sigma^{\text{multMatricesTr}|m_2} = \{mt\}$$

As with our other examples, both of the arguments are *used* but not *updated* and are therefore classified as *clean*. The sole operation, $(f \ m_2 \ r)$, is classified as *unobstructive*. Slicing for all arguments of f ,

$$\Sigma^{f|cs_0} = \{c, cs\} \quad \Sigma^{f|row} = \{row\}$$

we can see the same pattern: all the arguments are classified as *clean*, and the sole operation, $(\text{prodEscalar2} \ row \ c)$, is classified as *unobstructive*. Both `multMatricesTr`, and f can be rewritten as `map` operations.

We executed `matmult` for n between 1,000 and 5,000 at intervals of 500, with chunk size set to 20. Two parallel versions are possible: *i*) performs the multiplication of each row in parallel, and *ii*) divides each matrix into blocks to be computed in parallel and joins their result. We have used both modes here. Whilst the original *NoFib* suite definition provides its own matrix generation function, we have adjusted this so that matrices are generated using:

```
1 replicate n [1..n]
```

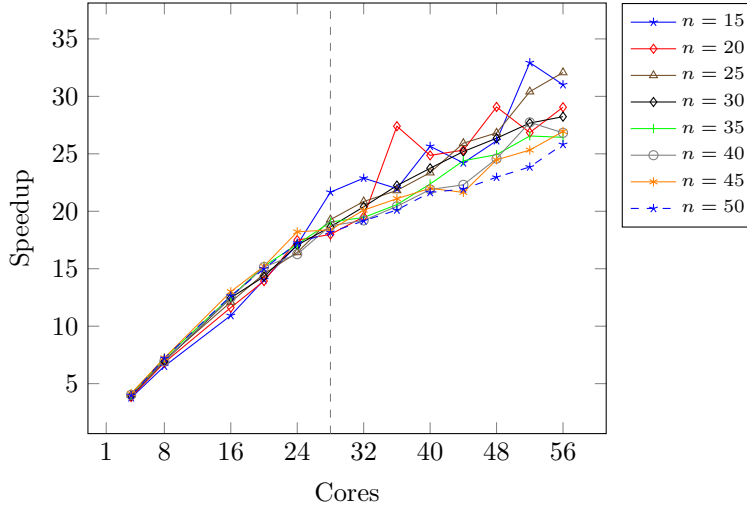


Figure 4.23: Speedups for `matmult` on *corryvreckan* using the block-wise parallelisation option and reported mutator (MUT) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores. n is a multiple of 10^2 .

since we discovered it produces better parallel performance. Figure 4.21 reports sequential times (with standard deviations), and the number of sparks, heap allocation size, and residency for each n . The average number of sparks converted varies with n , instead of the number of cores seen in previous examples. Converted sparks fluctuate between 100% and not less than 93% (seen on 4 cores), and fluctuations grow smaller with larger n . Figures 4.22, 4.24, 4.23, and 4.25 give speedups for `matmult` using mutator and total time for both parallel modes. Row-wise parallelisation achieves maximum speedups of 28.26 for $n = 2,500$ on 56 hyper-threaded cores for both mutator and total values. Block-wise parallelisation achieves maximum speedups of $32.93\times$ for $n = 1,500$ on 52 hyper-threaded cores for both mutator and total values. As with `nbody`, there is little difference between mutator and total reported speedups for `matmult`, and block-wise parallelisation proves generally only slightly better than row-wise parallelisation. The example demonstrates good scalability for both varying n and number of cores. Once again, speedup growth slows after 28 cores due to hyperthreading.

Interestingly, the definition of `f` might be unfolded (in the transformational sense) in `multMatricesTr`; e.g.

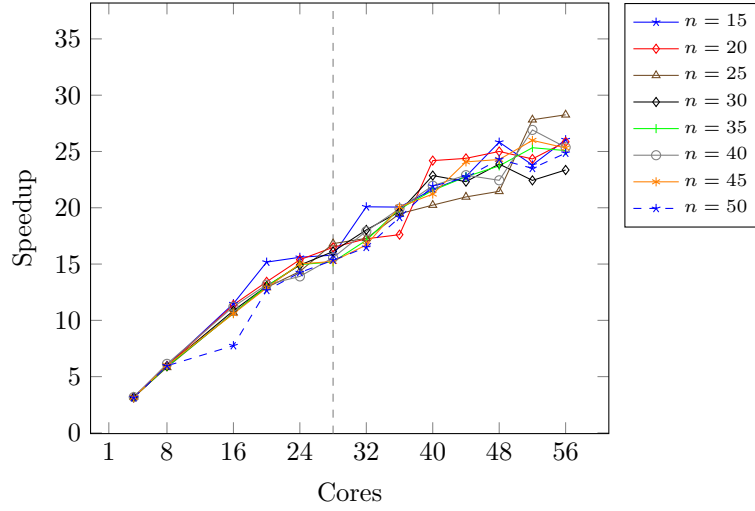


Figure 4.24: Speedups for `matmult` on *corryvreckan* using the row-wise parallelisation option and reported total (`Total`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores. n is a multiple of 10^2 .

```

1 multMatricesTr xs0@[] r m2 =
2   []
3 multMatricesTr xs0@(x:xs) rr@[] m2 =
4   multMatricesTr m2 x m2 : multMatricesTr xs [] m2
5 multMatricesTr xs0@(x:xs) rr m2 =
6   prodEscalar2 rr x : multMatricesTr xs rr m2

```

Here, `multMatricesTr` is now a fix-expression, case-splitting on its first argument, `xs0`. Slicing for its arguments, gives

$$\begin{aligned}
 \Sigma_{\text{multMatricesTr}|xs_0} &= \{x, xs, xs_0\} \\
 \Sigma_{\text{multMatricesTr}|rr} &= \{rr, \bar{r}r\} \\
 \Sigma_{\text{multMatricesTr}|m2} &= \{m2\}
 \end{aligned}$$

While `xs0` is considered to be updated, due to `m2` being passed as the first argument in the recursive call in line Line 4, both `xs0` and `m2` are considered to be *clean*. Conversely, `rr` is considered to be *tainted*. (`prodEscalar2 rr x`), now the sole operation, is therefore classified as *obstructive*, meaning that no map operations can be introduced. This is a correct result since `multMatricesTr` traverses *two* lists, where both the lists are passed as the first argument. The standard definition of `map`,

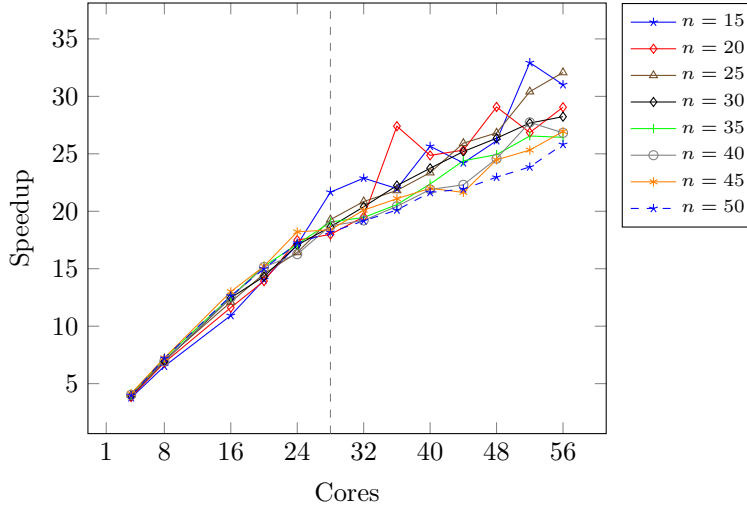


Figure 4.25: Speedups for `matmult` on *corryvreckan* using the block-wise parallelisation option and reported total (`Total`) time. Dashed line indicates 28 physical cores, with a total of 56 hyper-threaded cores. n is a multiple of 10^2 .

for example, does not allow this behaviour, and any attempt at introducing a `map` operation would result in a function that is not functionally equivalent to `multMatricesTr`.

Our translation of the `NoFib` benchmark code comprises 8 recursive functions over 2 Haskell modules. Within those 8 recursive functions there are 23 operations, of which 6 are classified as *unobstructive* and the remaining 17 are classified as *obstructive*.

Example	RFs	Ops	RCs	Args	Unobstructive Ops		Obstructive Ops		Time (μ s)	σ (μ s)
					Actual	Found	Actual	Found		
Data.List	18	22	20	31	5	5	17	17	6797.22	409.14
elem	1	2	1	2	1	1	1	1	176.28	26.51
list_ackermann	1	0	3	2	0	0	0	0	196.46	20.65
gcd	1	3	1	2	0	0	3	3	178.46	48.93
hanoi	1	5	1	1	0	0	5	5	156.16	39.50
k-means	4	15	5	14	4	4	11	11	646.30	60.90
quicksort	3	6	5	5	2	2	4	4	350.94	17.98
swaterman	2	10	2	10	0	0	10	10	420.92	18.44
sumeuler	14	47	14	25	16	16	31	31	2171.36	63.95
matmult	8	23	7	16	6	6	17	17	1195.62	63.23
queens	2	11	2	6	2	2	9	9	487.04	24.48
sudoku	5	9	5	8	3	3	6	6	519.66	38.68

Table 4.1: Examples run through slicing prototype implementation. Times are an average of 50 runs.

4.6.3 Other Examples

Our other examples, including the 18 functions from the Haskell Prelude `Data.List` library, may not all benefit from parallelisation, but serve as a demonstration of our approach. The 18 examples chosen from `Data.List` are: `and`, `or`, `append`, `foldl`, `foldr`, `init`, `intersperse`, `last`, `length`, `map`, `maximum`, `replicate`, `reverse`, `scan`, `heads`, `subsequences`, `tails` and `transpose`. These are a representative subset of the functions in the library; the remaining functions are similar to these. All our (translated) example code can be found at <https://adb23.host.cs.st-andrews.ac.uk/fhpc17-examples.zip>.

We give an overview of our results in Fig. 4.1, where `gcd` refers to the greatest common denominator function, and `hanoi` refers to the Towers of Hanoi puzzle. The `list_ackermann` example is a reimplementaion of the standard Ackermann function that traverses lists.

```
1 list_ackermann as0@[] bs0 = (1:bs0)
2 list_ackermann as0@(a:as) bs0@[] =
3   list_ackermann as [1]
4 list_ackermann as0@(a:as) bs0@(b:bs) =
5   list_ackermann as (list_ackermann a bs)
```

`k-means` refers to Lloyd’s K-Means algorithm and `swaterman` refers to the Smith-Waterman algorithm. Both are larger than the other examples, and we explore them in greater detail below.

Our results show that all operations are correctly classified. For each example, we give: the number of recursive functions that it contains (RFs), the total number of operations (Ops), the total number of recursive calls (RCs), the total number of arguments to all recursive functions (Args), the expected number of obstructive and unobstructive operations (Actual), those that are found by our prototype (Found), the average execution time for our prototype of 50 runs on *neptune* for that example (Time), and the standard deviation (σ) of those times.

As a synthetic benchmark, we have also applied our prototype to programs with varying input sizes, measured in the number of operations, by duplicating the translated `SumEuler` module m times. As the

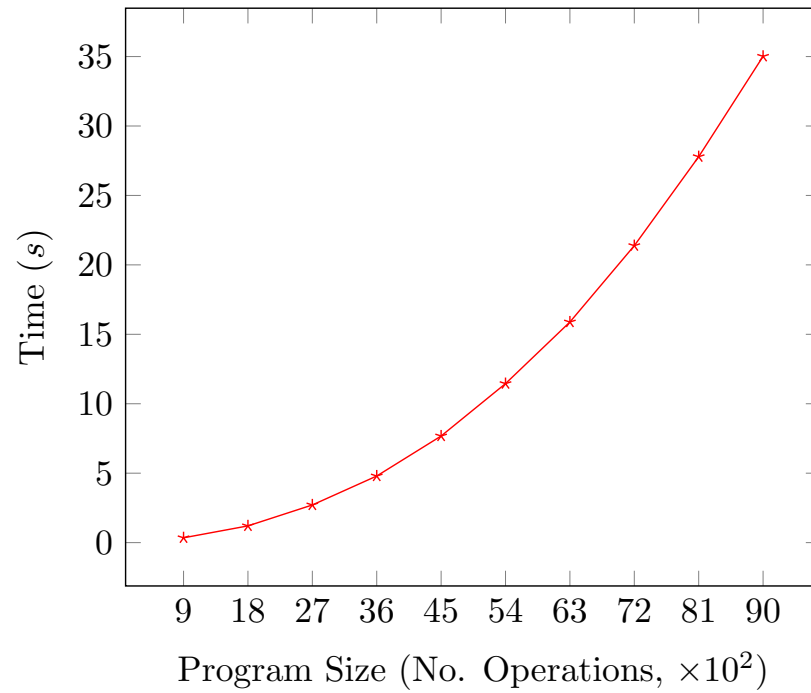


Figure 4.26: Prototype execution times for a program with varying sizes of input. Our implementation appears quadratic.

translated `SumEuler` has 9 operations, the total number of operations is $n = m \times 9$. Figure 4.26 shows the average time taken of five runs, in seconds, by our prototype on *neptune*. Our classifier takes a minimum of 0.35s for $n = 900$, with a standard deviation of 0.03s, and a maximum of 35.02s for $n = 9,000$, with a standard deviation of 0.58s. Our prototype implementation appears to in quadratic time with respect to n . The total time taken to classify each of our examples on *neptune* are all low, as shown in Figure 4.1. The longest our prototype takes to classify one of the examples in Figure 4.1 is 6.80ms.

K-Means

The goal of the K-Means problem is to partition a set of data points into clusters [86]. Lloyd’s algorithm finds a solution by iteratively improving upon an initial guess.


```

1 nearest p c cs0@[] = True
2 nearest p c cs0@(c':cs) =
3   if (dist c p) <= (dist c' p)
4   then nearest p c cs
5   else False
6
7 f n c cs ps0@[] r = div_p r n
8 f n c cs ps0@(p:ps) r =
9   if nearest p c cs
10  then f (n+1) c cs ps (add_p r p)
11  else f n c cs ps r
12
13 step cs' cs0@[] ps = []
14 step cs' cs0@(c:cs) ps =
15   (f 0 c cs' ps (origin 0)) : step cs' cs ps
16
17 loop 0 ps cs = cs
18 loop i ps cs = loop (i-1) ps (step cs cs ps)

```

Here, `loop` manages the total number of iterations, `step` updates each cluster point, `f` assigns points to a cluster to produce a new centroid, and `nearest` determines whether a point should be assigned to a cluster. To clarify our presentation, we omit the definitions of `dist` (calculates the distance between two Cartesian points), `add_p` (sums two points together), `div_p` (divides each part of a point by a scalar), and `origin` (produces the origin coordinate).

As in our quicksort example, we inspect each function in turn, with each function producing slices and *taintedness* classifications for each argument. Slicing `loop`,

$$\begin{aligned}
\Sigma^e|i &= \{i, \bar{i}\} && (\text{tainted}) \\
\Sigma^e|ps &= \emptyset && (\text{clean}) \\
\Sigma^e|cs &= \{cs, \bar{cs}\} && (\text{tainted})
\end{aligned}$$

`loop` has three operations: $(i-1)$, $(\text{step } cs \ cs \ ps)$, and the implicit if-expression and equality check represented by the pattern match on i .

All three operations are classified as *obstructive* since both i and cs are classified as *tainted*. We therefore find nothing to refactor in *loop*.

Slicing *step*,

$$\begin{aligned}\Sigma^e|_{cs'} &= \{cs'\} \quad (\text{clean}) \\ \Sigma^e|_{cs_0} &= \{c, cs\} \quad (\text{clean}) \\ \Sigma^e|_{ps} &= \{ps\} \quad (\text{clean})\end{aligned}$$

Since all arguments of *step* are classified as *clean*, all operations in *step* are considered to be *unobstructive*. The *origin* operation takes only a literal as argument, and the call to *f* is passed expressions that are literals, variables classified as *clean*, or operations that are classified as *unobstructive*. Since *step* is structurally recursive on cs_0 we can apply our refactoring defined in Section 4.5; the call to *f* is lifted into a *map* operation:

```
1 step cs' cs_0 ps =
2   step' cs' cs_0 ps
3   (map (\c -> f 0 c cs' ps (origin 0)) cs_0)
4
5 step' cs' cs_0@[] ps ds_0@[] = []
6 step' cs' cs_0@(c:cs) ps ds_0@(d:ds) =
7   d : step cs' cs ps ds
```

Since *step'* has no further operations, we might further rewrite *step* as a call to *map*:

```
1 step cs' cs_0 ps =
2   map (\c -> f 0 c cs' ps (origin 0)) cs_0)
```

The programmer might also choose to inline, perhaps using the *Inline Definition* refactoring, the call to *step* in *loop* on Line 18.

```
1 loop 0 ps cs = cs
2 loop i ps cs = loop (i-1) ps
3   (map (\c -> f 0 c cs' ps (origin 0)) cs)
```

Slicing f ,

$$\begin{aligned}\Sigma^{e|n} &= \{n, \bar{n}\} \quad (\text{tainted}) \\ \Sigma^{e|c} &= \{c\} \quad (\text{clean}) \\ \Sigma^{e|cs} &= \{cs\} \quad (\text{clean}) \\ \Sigma^{e|ps_0} &= \{ps\} \quad (\text{clean}) \\ \Sigma^{e|r} &= \{r, \bar{r}\} \quad (\text{tainted})\end{aligned}$$

f has five operations: *i)* $(n+1)$ on Line 10, *ii)* $(\text{add_p } r \text{ } p)$ also on Line 10, *iii)* $(\text{nearest } p \text{ } cs)$ on Line 9, *iv)* $\text{if} \dots \text{then} \dots \text{else}$ on Lines 9–11, and *v)* $\text{div_p } r \text{ } n$ on Line 7. Since n is classified as *tainted*, $(n+1)$ is considered to be *obstructive*. Similarly, since r is classified as *tainted*, both $\text{add_p } r \text{ } p$ ($\text{div_p } r \text{ } n$) are considered to be *obstructive*. Since, both p and cs are classified as *clean*, $(\text{nearest } p \text{ } cs)$ is considered to be *unobstructive*. Since the if-expression on Lines 9–11 has both *obstructive* operations and recursive calls as subexpressions to its arguments, it must be classified as *obstructive*. Since f is structurally recursive on ps_0 , we can apply our refactoring, so lifting $(\text{nearest } p \text{ } cs)$ in to a map operation:

```

1  f n c cs ps0 r =
2    f' n c cs ps0 r (map (\p -> nearest p c cs) ps0)
3
4  f' n c cs ps0@[ ] r qs0@[ ] = div_p r n
5  f' n c cs ps0@(p:ps) r qs0@(q:qs) =
6    if q
7    then f' (n+1) c cs ps (add_p r p) qs
8    else f' n c cs ps r qs

```

Since p occurs on Line 6 of f' we cannot remove ps_0 as an argument to f' . We additionally note that f is an instance of a *near fold*, specifically `foldl`. f could instead be rewritten as a composition of an uncurried `div_p` and a call to `foldl`, e.g.

```

1 f n c cs ps0 =
2   uncurry div_p
3   (foldl (\p (r,n) ->
4           if nearest p c cs
5           then (add_p r p, n+1)
6           else (r, n)) (r,n) ps0)

```

Since our approach is only able to detect `map` operations, we cannot rewrite `f` as a `foldl`.

Finally, we slice `nearest`,

$$\begin{aligned}
\Sigma^{e|p} &= \{p\} && (\text{clean}) \\
\Sigma^{e|c} &= \{c\} && (\text{clean}) \\
\Sigma^{e|cs_0} &= \{c', cs\} && (\text{clean})
\end{aligned}$$

`nearest` has three operations: *i*) (`dist c p`), *ii*) (`dist c' p`), and *iii*) (`...<=...`), all on Line 3. Since all arguments are classified as *clean*, all three operations are considered to be *unobstructive*. Moreover, since `nearest` itself is structurally inductive on `cs0`, we can apply our refactoring.

```

1 nearest p c cs0 = nearest' p c cs0
2   (map (\c' -> dist c p <= (dist c' p)) cs0)
3
4 nearest' p c cs0@[ ] ds0@[ ] = True
5 nearest' p c cs0@(c':cs) ds0@(d:ds) =
6   if d
7   then nearest p c cs ds
8   else False

```

By introducing the `map` operation, `nearest` will no longer *short-cut* upon failure as it did in its original definition. Similar functions, especially where the lifted operation is computationally expensive, may present a trade-off between efficiency gains by short-cutting operation, and performance improvements by parallelising the introduced `map`.

Smith-Waterman

The Smith-Waterman algorithm compares the similarity of two strings. The algorithm was originally designed for the comparison of nucleotides [108]. It has two stages: populating a matrix, and backtracking over the matrix to find the shortest ‘distance’ between the two strings. The code below concerns part of the matrix population stage. It traverses the matrix m , updating each cell with the result of h , where h (implementation omitted) calculates the maximum similarity score between the two strings a and b for the current row r and column c and updates the cell with that score.

```

1  tcs r c a b m =
2    if c > (length m)
3    then m
4    else tcs r (c+1) a b (h r c a b m)
5
6  trs r c a b m =
7    if r > (length m)
8    then m
9    else trs (r+1) c a b (tcs r c a b m)
10
11 traverse m a b = trs 1 1 a b m

```

As before, we calculate a slice for each argument to each function, classifying those arguments using the slice.

tcs:

$$\begin{aligned}
\Sigma^{tcs|r} &= \{r\} && \text{(clean)} \\
\Sigma^{tcs|c} &= \{c, \bar{c}\} && \text{(tainted)} \\
\Sigma^{tcs|a} &= \{a\} && \text{(clean)} \\
\Sigma^{tcs|b} &= \{b\} && \text{(clean)} \\
\Sigma^{tcs|m} &= \{m, \bar{m}\} && \text{(tainted)}
\end{aligned}$$

`trs`:

$$\begin{aligned}
 \Sigma^{\text{trs}|r} &= \{r, \bar{r}\} \quad (\text{tainted}) \\
 \Sigma^{\text{trs}|c} &= \{c\} \quad (\text{clean}) \\
 \Sigma^{\text{trs}|a} &= \{a\} \quad (\text{clean}) \\
 \Sigma^{\text{trs}|b} &= \{b\} \quad (\text{clean}) \\
 \Sigma^{\text{trs}|m} &= \{m, \bar{m}\} \quad (\text{tainted})
 \end{aligned}$$

Using these classifications, we can classify the operations within the recursive functions.

`tcs`: As both `c` and `m` are classified as *tainted*, all operations in `tcs` (i.e. `length`, and the less-than and addition operators) are classified to be *obstructive*.

`trs`: Analogous to `tcs`, and as both `r` and `m` are classified as *tainted*, all operations in `trs` are classified to be *obstructive*.

No refactoring can be applied to the Smith-Waterman implementation. However, the Smith-Waterman algorithm is an example of *wavefront parallelism* whereby cells of the matrix are divided into groups which can be calculated in parallel. As our classification looks only for independence across an entire data structure, at present this would be undetected.

4.7 Summary and Discussion

Skeleton-based approaches simplify the parallelisation process by abstracting over common patterns of parallelism and communication. Despite the advantages of this approach, the introduction and manipulation of skeletons, an often non-trivial task, is left to the programmer. Previous work has demonstrated that refactoring techniques can be used to simplify the introduction and manipulation of parallelism [11, 19, 43]. Knowing where and in what order to apply these refactorings becomes the next problem. While sequential higher-order functions, or recursion schemes, that have some parallel equivalent, e.g. `map`, can be used as loci for the potential introduction of parallelism, not all possible instances of, e.g., `map` are guaranteed to be found in code.

This chapter presents a novel program slicing-based approach to discover fully-applied application subexpressions, or *operations*, that may be performed as part of a `map` operation. In Section 4.1, we gave an overview of our approach using a simple Sudoku solver example. In section 4.2 we described the expression language used for our analysis, including assumptions and properties thereof. In Section 4.3 we presented the definitions and algorithms required to discern those operations that can be lifted into a `map` operation. This included formal definitions of *usage* and *update* for variables in our expression language (Section 4.3.1); a novel program slicing algorithm to determine whether a variable is used and updated in a given expression (Section 4.3.2); definitions for classifying variables as either tainted or clean according to a slice (Section 4.3.3); and finally, definitions for classifying operations as either obstructive or unobstructive in Section 4.3.4, indicating whether an operation may be lifted. Finally, in Section 4.5, we presented a composite refactoring that may lift unobstructive operations into `map` operations.

4.7.1 Limitations

As the approach presented in this chapter inspects operations within functions, functions can have largely arbitrary structures. Despite this, the way a function recurses can potentially obfuscate how variables are used and updated, and therefore limit the ability of our approach to accurately detect *unobstructive* operations. One example of this is mutual recursion. Consider, for example, the definition of `nqueens`, which solves how n queens can be placed on a chess board of size n^2 such that no queen may take another according to the standard rules of chess.

```
1 nqueens nq = gen nq 0 []
2
3 gen nq n b
4   | n >= nq = [b]
5   | otherwise = genloop nq n (gennext nq [b])
6
7 genloop nq n bs@[ ] = []
8 genloop nq n bs@(b:bs) = gen nq (n+1) b ++ genloop bs
```

Here, `nq` represents the number of queens; `gennext` finds the next safe position on a board to place a queen; `gen` controls the number of queens placed on each board; and `genloop` ensures that queens are placed on all boards. `gen` and `genloop` are *mutually recursive*. Inspecting both functions would result in `(gen nq (n+1) b)` in `genloop` (Line 8) being classified as unobstructive and subsequently lifted into a `map` operation.

```

1 genloop nq n bs0 =
2   genloop' (map (\b -> gen nq (n+1) b) bs0)
3
4 genloop' cs0@[] = []
5 genloop' cs0@(c:cs) = c ++ genloop' cs

```

Conversely, consider the definition of `nqueens` where `genloop` is *unfolded* (in the transformational sense) in `gen`.

```

1 nqueens nq = gen nq 0 [] (gennext nq [[]])
2
3 gen nq n b xs0@[] =
4   if n >= nq then [b] else []
5 gen nq n b xs0@(x:xs) =
6   gen nq (n+1) x (gennext nq [x]) ++ gen nq n b xs

```

Here, both `n` and `xs0` control the recursion, and with the exception of `nq`, all arguments are classified *tainted*. Given the slices,

$$\begin{aligned}
\Sigma^{\text{gen}|nq} &= \{nq\} \\
\Sigma^{\text{gen}|n} &= \{n, \bar{n}\} \\
\Sigma^{\text{gen}|b} &= \{b, \bar{b}\} \\
\Sigma^{\text{gen}|xs0} &= \{x, xs, x\bar{s}_0\}
\end{aligned}$$

`n`, `b`, and `xs0` are all *used and updated* between recursive calls. The recursive call in the first argument of `(++)` in Line 4 introduces an update for those three arguments between recursive calls. This results in all operations being classified as *obstructive*. This is a correct result because the same `xs0` is not traversed for all `n`. `gen` is a combination of two patterns: *iteration* and `map`; where our approach detects only `map` instances. Our inability

to discover combination patterns suggests that a prior normalisation stage, whilst not necessary, may be potentially useful in the discovery of potential `map` operations.

A related limitation arises in that, whilst data dependency information is used to calculate the slice, the slice cannot detect patterns within those dependencies. This means such patterns cannot be taken advantage of to, e.g., derive groups of independent elements within the traversed data structure. For example, consider the Smith-Waterman algorithm, which, originally designed for the comparison of nucleotides [108], compares the similarity of two strings. The algorithm has two stages:

1. populating a matrix, and
2. backtracking over the matrix to find the shortest ‘distance’ between the two strings.

Consider the matrix-population stage; specifically how the matrix is traversed as it is populated.

```
1 traverse m a b = trs 1 1 a b m
2
3 trs r c a b m =
4   if r > (length m)
5   then m
6   else trs (r+1) c a b (tcs r c a b m)
7
8 tcs r c a b m =
9   if c > (length m)
10  then m
11  else tcs r (c+1) a b (h r c a b m)
```

Here, a matrix `m` is traversed, updating each cell with the result of `h`, where `h` (implementation omitted) calculates the maximum similarity score between the two strings `a` and `b` for the current row `r` and column `c`. `m` is classified tainted in both `trs` and `tcs`, and `r` and `c` are classified tainted in `trs` and `tcs` respectively. All other variables are classified clean. All operations in `trs` and `tcs` are therefore classified obstructive,

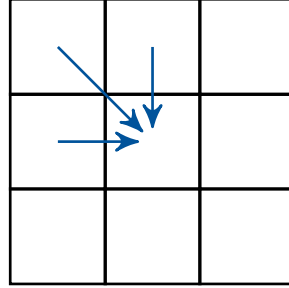


Figure 4.27: Data dependencies for each cell in the Smith-Waterman matrix.

meaning no refactoring can be applied to the Smith-Waterman implementation. However, the Smith-Waterman algorithm is an example of *wavefront parallelism* whereby cells of the matrix are divided into groups which can be calculated in parallel. In particular, the value of each cell is dependent upon three others, as shown in Figure 4.27, and so cells can be grouped into diagonal rows; e.g.

$$\begin{pmatrix}
 & - & a & c & a & c & a & c & t & a \\
 - & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
 a & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 g & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
 c & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
 a & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
 c & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
 a & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
 c & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
 a & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17
 \end{pmatrix}$$

where colour and number indicate groups, and ‘acacacta’ and ‘agcacaca’ are the strings being compared.

Another way in which the classification is found to be conservative is caused by the ‘Lamarckian inheritance’ of the classification of case-declared variables. Operations over the heads of tainted list variables may not themselves be obstructive. For example, consider \mathbf{f} ,

```

1 f n xs₀@[] = []
2 f n xs₀@(x:xs) = g x : f n (drop n xs₀)

```

where the operation $(g\ x)$ in Line 2 can be lifted into a map operation, and functional equivalence is maintained during refactoring to introduce the map operation. However, as xs_0 is classified tainted, $(g\ x)$ will be classified *obstructive*. This is a conservative result since there is no guarantee that xs_0 is updated in such a way that x is not dependent on previous recursive calls for all function definitions.

```

1 f g 0 xs₀ = xs₀
2 f g n xs₀@(x:xs) = g x : t g (n-1) (h xs₀)

```

Here, for example, xs_0 is updated in the recursive call via some function h . Suppose that we lift $(g\ x)$ in Line 2 into a map,

```

1 f g n xs = f' g n xs (map g xs)
2
3 f' g 0 xs₀ ys₀ = xs₀
4 f' g n xs₀@(x:xs) ys₀@(y:ys) =
5   y : f' g (n-1) (h xs₀) (h ys₀)

```

where we mimic the traversal of xs_0 in ys_0 . Should h change the values of the elements in xs_0 , and by extension the values of the elements in ys_0 , functional equivalence between the original and refactored definitions of f does *not* hold. We conjecture, based on experimental observations, that it is only when h changes the *values* of the elements in xs_0 that functional equivalence does not hold; conversely, when h *only* affects changes the *ordering* of elements or *size* of xs_0 , functional equivalence holds.

Finally, and in Section 4.5, where we define a refactoring to introduce a map operation, we observe that a separate refactoring is required for each definition of `map`. We further conjecture that should our technique be extended to allow arbitrary data types, and therefore arbitrary recursion schemes, e.g. `foldr`, separate refactorings will be required for each definition of each recursion scheme. This, in addition to needing to encode the behaviour of the recursion scheme to be detected in terms of variable usage and update, suggests the approach is relatively difficult to

extend. Relatedly, as this approach operates over an expression language, programs in languages such as Haskell or Erlang must be translated to the expression language for analysis. Alternatively, the expression language might be expanded to match the full syntax of, e.g., Haskell 98.

4.7.2 Concluding Remarks

In Chapter 3 we identified two key limitations of current approaches to automatic pattern discovery: *i)* the range of functions that can be inspected; and *ii)* the range of patterns that can be discovered. The first, limited range of functions, can lead to the introduction of runtime overheads, as in list homomorphism approaches [67], or require strict assumptions on the shape of the inspected function, as in hylomorphism approaches [30, 61]. Moreover, most current approaches inspect the function as a *whole* unit. Conversely, the approach described in this chapter is capable of both inspecting individual operations, and inspecting arbitrary recursive functions that are defined in our expression language. Inspecting individual operations allows us to discover patterns that could possibly have gone unnoticed by approaches that inspect the function as a whole. The approach by Ahn and Han [2], which also uses program slicing, is similar in this regard: they are able to find multiple pattern instances within a single function. Unlike our approach, however, which has no such limitation on the form of function inspected, they are only able to inspect functions that have a specific form and are first-order. We are therefore able to inspect a wider range of functions at the expression level when compared with existing approaches.

Despite our approach facilitating both discovery and introduction of patterns at the expression level and the analysis of arbitrary recursive functions, it is still only designed to discover `map` instances. The ability to discover multiple patterns is desirable since this enables both more *specific* and more *generic* patterns to be discovered, thereby enabling more of a program to be described in terms of recursion schemes, and thus more chances for parallelism. Moreover, a range of patterns leads to the greater likelihood that the behaviour of a program is more accurately captured by its recursion schemes. This can potentially lead to wider

opportunities for the kind of parallelism that is introduced without the need of further analysis, e.g. because a specific skeleton implementation requires a pattern that is not discoverable to be introduced, or the introduction of runtime overheads via, e.g., a projection function for an almost homomorphism [45].

In principle, it is possible to easily extend our approach to detect and introduce `zipWith` and `foldr/foldl` operations. Here, `zipWith` instances are a `map` operation that simultaneously recurses over two lists, and `foldr/foldl` instances can be derived from the function that remains from lifting *unobstructive* operations. Conversely, in order to discover additional patterns, the approach will likely require different definitions of obstructiveness in terms of *usage* (Definition 4.3.3) and *update* (Definition 4.3.1). It follows that, in order to discover multiple patterns, each operation must then be tested against the set of definitions of obstructiveness. However, in the case of new and programmer-defined patterns, the programmer must sufficiently understand and apply the concepts described in this chapter in order to define the pattern in terms of *usage* and *update* classifications. The programmer may even potentially redefine *usage* and/or *update* themselves should the current definitions prove to be too high-level to accurately describe the pattern's behaviour. An alternative approach to both the approach described in this chapter and to current automatic pattern discovery techniques, that is capable of discovering multiple kinds of patterns, but is also easily extensible, is therefore desirable. We describe one such approach in the following chapter.

Automatic Pattern Discovery via Anti-Unification

In this chapter, we describe a novel approach to discover and rewrite functions that may be expressed as a given higher-order function. This approach is designed to address the limitations identified in Section 4.7.2; specifically the discovery of multiple patterns. Here, we consider patterns to be pure Haskell 98 higher-order functions that are implementations of given *recursion schemes* [90]. In order to facilitate the discovery of multiple patterns, including patterns provided by the programmer, we use *anti-unification* [102, 105] to compare the structures of both the inspected function and pattern. This comparison allows us to determine whether the inspected function may be expressed as the pattern.

Section 5.1 gives an overview of the approach, including a simple example to illustrate it. Section 5.2 describes the preliminaries and assumptions that are required for the technique to work. Section 5.3 defines a novel anti-unification algorithm, and Section 5.4 defines properties that determine whether the inspected function is an instance of the given recursion scheme using the results of the anti-unification. Section 5.4.1 describes how the inspected function is rewritten as a call to the higher-order function. Section 5.5 describes how we implemented our approach using the Haskell refactoring tool, HaRe. Section 5.6 describes how the approach can be modified to discover `unfold` instances, where a data structure is constructed *corecursively*. Finally, Section 5.8 gives an

overview of the approach that is presented in this chapter, discussing strengths and weaknesses, and compares the approach against the previous chapter. The content of this chapter was presented at Lambda Days 2017, and accepted for publication in Future Generation Computer Systems [9].

5.1 Introduction

We illustrate our approach using an example from the standard *NoFib* suite of Haskell benchmarks [99], `sumeuler`, that calculates Euler's totient function for a list of integers and sums the results.

```
1 sumeuler :: [Int] -> Int
2 sumeuler xs = sum (map euler xs)
```

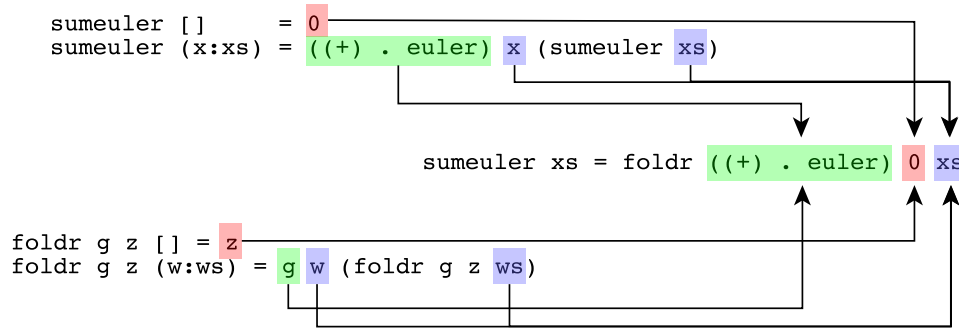
Here, `euler` is Euler's totient function, and `sum` sums the values in its argument, a list of integers. As demonstrated in Section 4.6.2, `sumeuler` can be easily parallelised using, e.g., the Haskell Strategies library:

```
1 sumeuler  :: [[Int]] -> Int
2 sumeuler xs =
3   sum (map (sum . map euler) xs)
4           `using` parList rdeepseq)
```

The sequential definition of `sumeuler` may be unfolded (in the transformational sense) in a number of ways. One approach is to fuse the calls to `sum` and `map` to produce a call to a `foldr`.

```
1 sumeuler :: [Int] -> Int
2 sumeuler xs = foldr ((+) . euler) 0 xs
3   where foldr g q [] = q
4           foldr g q (w:ws) = g w (foldr g q ws)
```

This new definition of `sumeuler` can then be parallelised in an alternative way to the parallel map above, using e.g. a *reduce* skeleton. As before, taking advantage of parallelism in `sumeuler` is therefore both *simple* and potentially *automatic* [12, 31, 107]. However, if `sumeuler` was defined

Figure 5.1: Rewriting `sumeuler` as an instance of `foldr`

without using either an explicit `map` or `foldr`, e.g. as shown below using direct recursion, then parallelisation could be less straightforward, since the computations that can be done in parallel may not be obvious; and lower, or no, speedups might result.

```

1 sumeuler :: [Int] -> Int
2 sumeuler [] = 0
3 sumeuler (x:xs) = ((+) . euler) x (sumeuler xs)

```

Since this version of `sumeuler` is *implicitly* an instance of `foldr`, it is best to first restructure the definition so that it *explicitly* calls `foldr` before attempting parallelisation. In order to rewrite the inlined `sumeuler` as an explicit `foldr`, however, we need to derive concrete arguments to `foldr` that will yield a functionally equivalent definition. These arguments can be derived by inspecting the inlined definition of `sumeuler`, as shown in Figure 5.1. To *automatically* derive the arguments to `foldr`, we inspect the definitions of `sumeuler` and `foldr` using *anti-unification*, which aims to find the *least general generalisation* between two terms. In Plotkin and Reynolds' original work [102, 105], anti-unification was defined for totally ordered terms, where terms consisted of variables, literals, and function application. More recent approaches to anti-unification have applied the technique to real programming languages such as Haskell [23, 24], primarily for clone detection and elimination [25]. In these approaches, anti-unification compares two terms (expressions) to find their shared structure, producing an *anti-unifier* term (representing the shared struc-

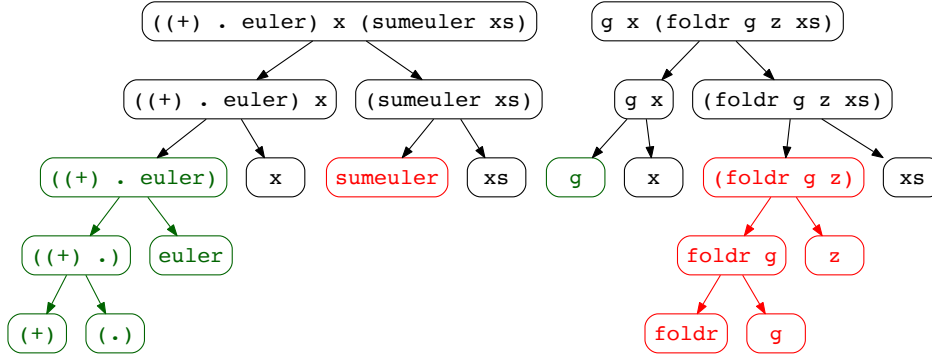


Figure 5.2: ASTs for the `(:)` clause of `sumeuler` (left) and `foldr` (right); shared structure in black, differing structure in green and red.

ture), plus two sets of *substitutions* that enable the original term to be reconstructed from the anti-unifier. For example, anti-unifying the `[]` clause of `sumeuler` with `foldr` compares the syntactically unequal `0` and `z` expressions, producing the anti-unifier, h :

```
1 h g z [] = z
```

Where the anti-unified structures diverge, a variable, referred to as a *hedge variable*, is introduced in the anti-unifier that can be substituted for the original term. Substitutions, σ , have the form $(t_i \mapsto t_j)$, where the t_i, t_j are terms. Substitutions are applied to terms, written using postfix notation, e.g. $t \sigma$, and can be composed like functions. An applied substitution replaces all instances of t_i with t_j in t . For example, the substitutions for the `[]` clause of `sumeuler` are:

$$\sigma_1 = (z \mapsto 0)$$

and the substitutions for the `[]` clause of `foldr` are ε , i.e. the identity operator for substitutions. Anti-unifying the respective `(:)` clauses of `sumeuler` and `foldr` produces the anti-unifier:

```
1 h g z (x:xs) = g x (\alpha xs)
```

where α is a free variable. As shown in Figure 5.2, the structures of the two clauses are very similar, consisting primarily of application expressions. Differences can be found at the leaves of `foldr`, highlighted

in both dark green and red in Figure 5.2. Since x and xs are in the same relative positions, they feature in h . As $((+) . \text{euler}) \neq g$ (highlighted in dark green), g is used to represent the function applied to two arguments in h . Finally, since the recursive call prefixes `sumeuler` and `(foldr g z)` (highlighted in red) are not syntactically equal, the variable α is used in their place in h . This produces the substitutions: $(g \mapsto ((+) . \text{euler}))$ and $(\alpha \mapsto \text{sumeuler})$ for `sumeuler`, and: $(\alpha \mapsto (\text{foldr g z}))$ for `foldr`. As shown in Figure 5.2, the structures of the two `cons (:)` clauses are very similar, consisting primarily of application expressions. Differences can be found at the leaves of `foldr`, with corresponding differences highlighted in both green and red across functions in Figure 5.2. Since x and xs are in the same relative positions, they feature in h . As $((+) . \text{euler})$ is not syntactically equal to g (highlighted in green), g is used to represent the function applied to two arguments in h . Finally, since `sumeuler` is not syntactically equal to `foldr g z` (highlighted in red), the variable a is used in their place in h . This produces the substitutions:

$$\begin{aligned}\sigma_1^2 &= (g \mapsto ((+) . \text{euler})) \circ (a \mapsto \text{sumeuler}) \\ \sigma_2^2 &= (a \mapsto \text{foldr g z})\end{aligned}$$

for `sumeuler` and `foldr`, respectively. Finally, since it is possible to reconstruct the original terms by applying the relevant substitutions to an anti-unifier, when given the anti-unifier h

```
1 h g z [] = z
2 h g z (x:xs) = g x (a xs)
```

we can, in principle, rewrite `sumeuler` and `foldr` in terms of h using substitutions as arguments:

```
1 sumeuler xs = h ((+) . euler) 0 xs
2
3 foldr g z xs = h g z xs
```

Furthermore, because `foldr` is *equivalent* to h , we conclude that `sumeuler` must be an instance of `foldr` and can be rewritten in terms of h , it must

be the case that `sumeuler` can be rewritten in terms of `foldr`. The substitutions for `sumeuler` inferred as part of anti-unification are valid as arguments to `foldr`, allowing `sumeuler` to be rewritten:

```
1 sumeuler xs = foldr ((+) . euler) 0 xs
```

Parallelism can now be introduced using a parallel implementation of `foldr`, either manually or using a refactoring/rewriting tool (e.g. [17, 22]). Alternatively, the `foldr` operation can be split into its `map` and `foldr (sum)` components, perhaps by using the laws of hylomorphisms as in [31], producing:

```
1 sumeuler xs = foldr (+) 0 (map euler xs)
```

which is equivalent to our original definition of `sumeuler`:

```
1 sumeuler xs = sum (map euler xs)
```

This can then be parallelised using the parallel map that we originally showed.

The below sections give an overview of our approach. Figure 5.3 provides a key to our notation. Some function f that matches a pattern, p , in the set of patterns, \mathcal{P} , is rewritten to a new function, f' , that calls p and that is functionally equivalent to f . Our approach determines whether f is an instance of p , and if so, derives the arguments that are needed to make f and f' functionally equivalent. This is achieved in two main stages:

1. anti-unifying f and p to derive argument candidates; and
2. using the result of the anti-unification to determine if f is an instance of p , and if so, validating the argument candidates.

Our approach compares two functions, and given a finite set of \mathcal{P} , can be applied repeatedly to discover a set of potential rewrites. When multiple rewrites are valid, we take the first, although other selection methods are also valid.

\mathcal{P}	=	Set of pattern implementations
f	=	Function to be transformed
p	=	A function in \mathcal{P}
h	=	Result of the anti-unification of f and p
f'	=	f rewritten as an instance of p
a_i	=	Argument of f or p being recursed over
v_{a_i}	=	Variable declared in a_i
t, t_i	=	Terms for anti-unification
σ, σ_i	=	Substitutions for anti-unification
ε	=	The identity substitution
α	=	Hedge variables, found only in h

Figure 5.3: Key to terms and notation.

5.2 Preliminaries and Assumptions

We illustrate our approach using the Programatica AST representation of the Haskell 98 standard [104]. Programatica is used as a basis for the Haskell refactoring tool, HaRe, which we extend to implement our approach. We discuss our implementation in Section 5.5. Programatica represents a Haskell AST using a collection of 20 Haskell data types, comprising a total of 110 constructors [18]. These types are parameterised by the location of the syntactic elements they represent, and in the case of variables and constructors, where they are declared. For example, the expression 42 is represented in Programatica by `(Exp (HsLit loc (HsInt 42)))`. Here, `HsLit` indicates that 42 is an expression; `HsInt` indicates that it is an integer; and `loc` represents its location in the source code. We refer to the representation of Haskell *expressions* in Programatica as *terms*; i.e. given some expression e and some term t , we say that t is the Programatica representation of e (denoted $\llbracket e \rrbracket = t$). In the above example, $\llbracket 42 \rrbracket = (\text{Exp } (\text{HsLit } \text{loc } (\text{HsInt } 42)))$; i.e. `(Exp (HsLit loc (HsInt 42)))` is the *term* that represents the *expression* 42. There is a one-to-one mapping between expressions and terms. In the Programatica tool set, terms are values of the `HsExpI` type. We make the distinction between expressions and terms since terms can be easily generalised over using their constructors, as we do in Section 5.3.

Although in principle we anti-unify the Haskell functions f and p , we

do so in a structured way that we define in Section 5.3. Beyond this, we do not need to consider (the representations of) arbitrary Haskell declarations or modules since we only anti-unify terms that form the right-hand side of like-equations in \mathfrak{f} and \mathfrak{p} . Since location information will mean that any two compared terms will *always* be unequal, we *discard location information*. Finally, and to simplify our presentation, we will omit the outer `Exp` constructor, the common `Hs` prefix of constructors, and any explicit specification of literal terms (e.g. integer or string). For example, `(Exp (HsLit loc (HsInt 42)))` is instead recorded as `(Lit 42)`. Where variables and constructors are both represented as identifiers, we will instead record these using `Var` and `Con`, respectively. For example, the term of the variable expression `x` is `(Var x)`, and the term of the *cons* operator expression, `(:)`, is `(Con (:))`. While our approach works for all terms, here we only need to explicitly refer to: `Var` (representing variables), `Lit` (representing literals), and `App` (representing application expressions). All other terms fall under the generic constructor `C`, and lists of terms (e.g. `[t1, ..., tn]`) that used as arguments to `C` to represent lists and tuples. Other common expressions found in functional languages, such as lambdas, fall under the general case since they are both unlikely to appear in recursion scheme implementations, and are otherwise safe to anti-unify due to the assumptions below.

5.2.1 Assumptions

In order to determine whether \mathfrak{f} is an instance of \mathfrak{p} , we will assume:

1. that variables are unique in \mathfrak{f} and \mathfrak{p} ;
2. that no variables are free in \mathfrak{p} , and that all variables in \mathfrak{p} are declared as arguments to \mathfrak{p} ;
3. that \mathfrak{f} and \mathfrak{p} recurse on the same type (denoted τ);
4. that \mathfrak{f} and \mathfrak{p} only pattern match on the argument that is being traversed (denoted a_i , where a_i is the i^{th} argument to \mathfrak{f} and/or \mathfrak{p});

5. that for every clause that matches a constructor of τ in p , there is a corresponding clause that matches the same constructor in f , and vice versa; and
6. that no parameter to a_i (denoted v_{a_i}) occurs as part of any binding in an as-pattern, let-expression, or where-block.

Clauses in p are always anti-unified with like clauses in f , according to the constructor of τ matched for that clause, C . a_i then acts as ‘common ground’ between f and p ; i.e. both functions declare the parameters of the constructor in a_i as arguments. This enables a check to ensure that the inferred substitutions are *sensible* (Definition 5.4.6). It is therefore useful to know which v_{a_i} in f correspond to which v_{a_i} in p . Despite this, since variables in f and p are assumed to be unique, it follows that no variable term in f will ever be syntactically equal to any variable term in p . We instead consider v_{a_i} with the same position in f and p to be equivalent.

Definition 5.2.1 (Shared Argument Equivalence). *Given the argument of type τ that is pattern matched in both f and p , a_i , where in f , $a_i = (D v_1 \dots v_n)$, and in p , $a_i = (D w_1 \dots w_n)$, we say that each v_i is equivalent to each w_i (denoted $v_i \equiv w_i$); i.e. $\forall i \in [1, n], v_i \equiv w_i$. Given two arbitrary v_{a_i} in f and p , $v_{a_i}^f$ and $v_{a_i}^p$, we denote their equivalence by $v_{a_i}^f \equiv v_{a_i}^p$.*

In `suneuler` and `foldr` from Section 5.1, for example, `x` and `xs` are v_{a_i} in `suneuler`, and `w` and `ws` are similarly v_{a_i} in `foldr`. Here, both $x \equiv w$ and $xs \equiv ws$ hold, since `x` and `w` are the first arguments to their respective *cons* operations, and `xs` and `ws` are the second arguments.

Traditional anti-unification algorithms use syntactic equality to relate two terms, t_1 and t_2 , their anti-unifier, t , and the substitutions σ_1 and σ_2 . To take advantage of argument equivalence we must use a weaker form of syntactic equality. First, we define the binary relation \sim over terms, a form of alpha equivalence.

Definition 5.2.2 (Alpha Equivalent Terms). *Given two variables, $v_{a_i}^f$ and $v_{a_i}^p$, we say that the term representations of $v_{a_i}^f$ and $v_{a_i}^p$ are equivalent (denoted $\llbracket v_{a_i}^f \rrbracket \sim \llbracket v_{a_i}^p \rrbracket$) when:*

$$\frac{v_{a_i}^f \equiv v_{a_i}^p}{\text{Var } v_{a_i}^f \sim \text{Var } v_{a_i}^p}$$

We then replace syntactic equality with a weaker form of equivalence using Definition 5.2.2.

Definition 5.2.3 (Syntactic Equivalence). *We define syntactic equivalence to be a binary relation over two terms, t_1 and t_2 , denoted $t_1 \cong t_2$, where \cong is the reflexive structural closure of \sim .*

For example, $(\text{Lit } 42) \cong (\text{Lit } 42)$ holds; $t_1 \cong t_2$ holds for all $t_1 = (C\ t_{11} \dots t_{1n})$ and $t_2 = (C\ t_{21} \dots t_{2n})$ when $\forall i \in [1, n], t_{1i} \cong t_{2i}$; and finally, $(\text{Var } v_{a_i}^f) \cong (\text{Var } v_{a_i}^p)$ holds *only* when $v_{a_i}^f \equiv v_{a_i}^p$ is true. We note that for all other variables, v, w , $(\text{Var } v) \cong (\text{Var } w)$ *does not hold*.

Where τ is a product type, e.g. as in `zipWith`, we permit pattern matching on all arguments that represent the data structure(s) being traversed. All other arguments must be declared as simple variables. For example, given:

```
1 f1 a b c [] [] = []
2
3 f2 0 (b',b'') c [] = []
```

`f1` is permitted, but `f2` is not.

As-patterns, `let`-expressions, and `where`-blocks all enable occurrences of some v_{a_i} to be aliased, or *obfuscated*, and thereby potentially obstruct argument derivation. Since our analysis inspects the syntax of f and p , such patterns and expressions can obfuscate the exact structure of f ; i.e. lift potentially important information out of the main body of f and p . For example, given the definition,

```
1 f3 [] = []
2 f3 (x:xs) = g1 x : f3 xs where g1 = g2 xs
```

the fact that `xs` is passed to `g2` is obfuscated by the `where`-block, and could lead to an incorrect rewriting of `f3` as, e.g., a `map`. These syntactic constructs could be removed (semi-)automatically prior to analysis.

We do not restrict the type of recursion; general recursive forms are allowed, for example. Partial definitions are also allowed. For example, both `zipWith` and `zipWith1` are valid as implementations of the general *zipWith* recursion scheme:

```

1 zipWith g [] [] = []
2 zipWith g (x:xs) (y:ys) = g x y : zipWith xs ys
3
4 zipWith1 g1 g2 g3 [] [] = []
5 zipWith1 g1 g2 g3 (x:xs) [] = g1 x : zipWith1 xs []
6 zipWith1 g1 g2 g3 [] (y:ys) =
7   g2 y : zipWith1 [] ys
8 zipWith1 g1 g2 g3 (x:xs) (y:ys) =
9   g3 x y : zipWith1 xs ys

```

This permits more implementations of a given scheme, and so increases the likelihood of discovering an instance of a scheme in f .

Finally, and since our approach effectively requires that f has a similar syntactic structure to p as a result of the anti-unification, we permit any valid and finite normalisation procedure that rewrites some arbitrary f_0 to f , such that f_0 is *functionally equivalent* to f . Normalisation can be used, e.g., to ensure that any of the above assumptions are met. For example, consider assumption 6: as-patterns and definitions in `let`-expressions and `where`-blocks can be inlined (or *unfolded* in the transformational sense). For example, the `where`-block definition of `g1` in f_3 can be unfolded to produce:

```

1 f3 [] = []
2 f3 (x:xs) = g2 xs x : f3 xs

```

Alternatively, normalisation procedures can be used to *reshape* [11] functions into a form to allow, or simplify, the discovery of recursion schemes. For example, the definition,

```

1 f4 xs0 = case xs0 of
2         [] -> 0
3         (x:xs) -> x + f xs

```

can be rewritten to lift the case-split into a pattern match, allowing the function to be anti-unified against a `foldr`:

```

1 f4 [] = 0
2 f4 (x:xs) = x + f xs

```



```

3
4 foldr g z [] = z
5 foldr g z (x:xs) = g x (foldr g z xs)

```

More ambitious normalisation procedures may split a function, e.g. `f5`:

```

1 f5 [] z = g2 z
2 f5 (x:xs) z = f5 xs (g1 x z)

```

which can be considered a *near fold*. Here, we can lift out the application of `g2` in the base case, and therefore expose a `foldl` instance:

```

1 f5 xs z = g2 (f5' xs z)
2
3 f5' [] z = z
4 f5' (x:xs) z = f5' xs (g1 x z)

```

Splitting of functions in this way is allowed provided that any such splitting is *finite* and *reduces the size of the split function*. For example, some `f0` cannot be split into the composition of `f0'` and `f0''`, where `f0'` or `f0''` is functionally equivalent to the identity operation. In line with the intention for `f` to be anti-unified against a set of recursion schemes, we do *not* require the normalisation procedure to be confluent.

The normalisations discussed above can be performed automatically, semi-automatically via a refactoring process, or manually. A manual approach is not desirable, since it is prone to the introduction of errors, but is primarily used in the below chapter due to time constraints. A list of normalisations that we applied to our examples is given in Section 5.7.1. We conjecture that a small group of common normalisations, e.g. lifting as-patterns, alpha-renaming for uniqueness, and splitting or joining clauses, could facilitate normalisation for a significant range of functions. Implementation of these normalisations is left to future work.

5.3 Argument Derivation via Anti-Unification

In order to derive arguments to express `f` as an instance of `p`, we must anti-unify `f` and `p`. We define anti-unification for two levels: *i*) at the

term level for two arbitrary terms; and *ii*) at the function level where we anti-unify \mathfrak{f} and \mathfrak{p} . At the term level, the anti-unification of two terms, t_1 and t_2 , (denoted $t_1 \triangleq t_2$) we obtain the triple $(t \times \sigma_1 \times \sigma_2)$, where t is the anti-unifier with respect to the substitutions σ_1 and σ_2 . At the function level, the anti-unification of \mathfrak{f} and \mathfrak{p} produces a list of triples that represents the results of anti-unification for each pair of clauses in \mathfrak{f} and \mathfrak{p} .

We define a *substitution* to be a function that takes a variable, x , and returns a term, t_2 . When applied to a term, t_1 , a substitution returns t_1 but with all occurrences of x replaced by t_2 .

Definition 5.3.1 (Substitution). *Given two terms, t_1, t_2 , and a variable name, x , that occurs in t_1 , substituting x for t_2 in t_1 replaces all occurrences of $(\text{Var } x)$ in t_1 with t_2 . This substitution is denoted $(x \mapsto t_2)$; and the application of substitutions to terms is denoted $t_1 (x \mapsto t_2)$, where $(x \mapsto t_2)$ is applied to t_1 . We use σ to refer to substitutions; e.g. $\sigma = (x \mapsto t_2)$.*

Here, we update the classical definition of substitution [105] to account for the Programatica representation. Substitutions are denoted as substituting a variable *name* for some term. When applied to a term, a substitution replaces all *variable terms that are parameterised by the given variable name* with another term. For example, the result of the substitution σ applied the term t , $t\sigma$, is:

$$\begin{aligned} t &= \text{Var } g \\ \sigma &= (g \mapsto (\text{App } (\text{App } (\text{Var } .) (\text{Var } +)) (\text{Var } \text{euler}))) \\ t\sigma &= (\text{App } (\text{App } (\text{Var } .) (\text{Var } +)) (\text{Var } \text{euler})) \end{aligned}$$

Here, g is substituted for a term representing the expression $((+) . \text{euler})$, from `sumeuler`. While the *name* of the variable being substituted is given in σ , it is the variable term parameterised by g , i.e. $(\text{Var } g)$, in t that is substituted. We call the variables in substitutions that are substituted for terms *hedge variables*. Uniqueness of variables means that all occurrences of a hedge variable in a term can be substituted safely.

Definition 5.3.2 (Identity Substitution). *The identity substitution, denoted ε , is defined such that for all terms t , $t = t\varepsilon$.*

Substitutions can be composed using an n-ary relation such that composition forms a rose tree of substitutions.

Definition 5.3.3 (Substitution Composition). *For all terms t_0, \dots, t_n , and for all substitutions $\sigma_1, \dots, \sigma_n$, where $t_0 = (C \ t_1 \ \dots \ t_n)$, there exists some substitution $\sigma_0 = \langle \sigma_1, \dots, \sigma_n \rangle$ such that $t_0 \sigma_0 = (C \ (t_1 \sigma_1) \ \dots \ (t_n \sigma_n))$. Similarly, where $t_0 = [t_1, \dots, t_n]$, there exists some substitution $\sigma_0 = \langle \sigma_1, \dots, \sigma_n \rangle$ such that $t_0 \sigma_0 = [(t_1 \sigma_1), \dots, (t_n \sigma_n)]$.*

For the function f , defined

$$\begin{aligned} f \ v_{11} \ \dots \ v_{1n} &= e_1 \\ &\vdots \\ f \ v_{21} \ \dots \ v_{2n} &= e_n \end{aligned}$$

and for the substitutions $\sigma_1, \dots, \sigma_n$, there exists some substitution

$$\sigma_0 = \langle \sigma_1, \dots, \sigma_n \rangle$$

such that $\llbracket f \rrbracket \sigma_0$ is defined:

$$\begin{aligned} f \ v_{11} \ \dots \ v_{1n} &= e'_1 \\ &\vdots \\ f \ v_{21} \ \dots \ v_{2n} &= e'_n \end{aligned}$$

where

$$\forall i \in [1, n], \llbracket e'_i \rrbracket = \llbracket e_i \rrbracket \sigma_i$$

Composed substitutions are *ordered*, such that when applied to some term t_0 , the first composed substitution is applied to the first direct subterm to t_0 , the second composed substitution is applied to the second direct subterm, and so on. Each composed substitution is only applied to its corresponding direct subterm in t_0 , and does not interfere with any other subterm of t_0 . For example, the expression $(g \ x \ x)$ is represented by the term t_0 ,

$$t_0 = \text{App} (\text{App} (\text{Var } g) (\text{Var } x)) (\text{Var } x)$$

and given the substitution, σ_0 ,

$$\sigma_0 = \langle \langle (g \mapsto \text{Var } \text{euler}), \varepsilon \rangle, (x \mapsto \text{Var } p) \rangle$$

when we apply σ_0 to t_0 , the resulting term is:

$$t_0 \sigma_0 = \text{App} (\text{App} (\text{Var euler}) (\text{Var x})) (\text{Var p})$$

A composition of substitutions may be applied to either f or p , where each substitution applies to the right-hand side of each equation in order. We will refer to such compositions by σ_f and σ_p . For example, given the function `sum`,

```
1 sum [] = []
2 sum (x:xs) = x + (sum xs)
```

and the substitution $\sigma_f = \langle \sigma_1, \sigma_2 \rangle$, applying σ_f to `sum`, applies σ_1 to the term representation of `[]`, i.e. the right-hand side of the first equation, and σ_2 to the term representation of `(x + (sum xs))`, i.e. the right-hand side of the second equation.

Anti-unification computes the *anti-unifier* of two terms. In the literature, all hedge variables are *fresh*. Conversely, and since we aim to rewrite f in terms of p , we can use the variables declared in p as hedge variables. This allows us to easily derive expressions that can be passed to p in f . We refer to the set of hedge variables from p by \mathcal{V}_p . All hedge variables not in \mathcal{V}_p are fresh hedge variables, which we denote by α .

Another difference to traditional anti-unification algorithms can be found the concept of *unobstructive recursive prefixes*. As we are anti-unifying two different recursive functions, by definition they will diverge in at least one respect: their recursive calls. We extend this notion to *recursive prefixes*, where a recursive prefix is a variable expression comprising the name of the function, i.e. f or p , or an application expression that applies f or p to one or more expressions. For example, given the definition of `foldr`,

```
1 foldr g z [] = z
2 foldr g z (x:xs) = g x (foldr g z xs)
```

the recursive call, `(foldr g z xs)`, has four recursive prefixes, including: *i)* `(foldr g z xs)`; *ii)* `(foldr g z)`; *iii)* `(foldr g)`; and *iv)* `foldr`. It is useful to further extend the notion of recursive prefixes

with the concept of *unobstructiveness*. We define an *unobstructive recursive prefix* to be a recursive prefix for which the values of its arguments *do not change*. In the above example, all recursive prefixes, with the exception of $(\text{foldr } g \ z \ xs)$, are *unobstructive*.

Definition 5.3.4 (Unobstructive Recursive Prefix). *Given a recursive function f ,*

$$f \ v_1 \ \dots \ v_n = e$$

for all recursive calls in e ,

$$f \ e_1 \ \dots \ e_n$$

we say that a recursive prefix is either:

1. *the direct subexpression $f \ e_1 \ \dots \ e_m$, where $1 \leq m \leq n$; or*
2. *the expression f itself.*

We say that a recursive prefix is unobstructive when $\forall i \in [1, m], e_i = v_i$. An expression, e' is denoted as being an unobstructive recursive prefix by membership of the set of unobstructive recursive prefixes for f ; i.e. $e' \in \mathcal{R}_f$.

To help ensure that as many instances of some scheme are found, we will also assume the existence of the identity operator, which is defined in Haskell as `id`.

Definition 5.3.5 (The Identity Operation). *We define id to be the lambda expression $(\backslash x \rightarrow x)$ such that for all terms, t , $(\text{App } (\text{Var } id) (t)) \cong t$.*

Our anti-unification algorithm calculates the anti-unifier t and substitutions σ_1 and σ_2 for the terms t_1 and t_2 , such that $t_1 \cong t \sigma_1$ and $t_2 \cong t \sigma_2$ hold, denoted:

$$t_1 \triangleq t_2 = t \times \sigma_1 \times \sigma_2$$

Our rules in Figure 5.4 define how to infer t , σ_1 , and σ_2 from t_1 and t_2 . Rules have the form,

$$\frac{p}{\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (t) \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}}$$

$$\begin{array}{c}
 \text{EQ} \frac{t_1 \cong t_2}{\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (t_2) \begin{bmatrix} \varepsilon \\ \varepsilon \end{bmatrix}} \\
 \\
 \text{VAR} \frac{v \neq p}{\begin{bmatrix} t_1 \\ \text{Var } v \end{bmatrix} \cong (\text{Var } v) \begin{bmatrix} (v \mapsto t_1) \\ \varepsilon \end{bmatrix}} \\
 \\
 \text{ID} \frac{u \neq p \quad v_{a_i}^1 \equiv v_{a_i}^2}{\begin{bmatrix} \text{Var } v_{a_i}^f \\ \text{App } (\text{Var } u) (\text{Var } v_{a_i}^p) \end{bmatrix} \cong (\text{App } (\text{Var } u) (\text{Var } v_{a_i}^p)) \begin{bmatrix} \langle (u \mapsto id), \varepsilon \rangle \\ \varepsilon \end{bmatrix}} \\
 \\
 \text{RP} \frac{t_2 \in \mathcal{R}_p}{\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (\text{Var } \alpha) \begin{bmatrix} (\alpha \mapsto t_1) \\ (\alpha \mapsto t_2) \end{bmatrix}} \\
 \\
 \text{CONST} \frac{\forall i \in [1, n], \begin{bmatrix} t_{1i} \\ t_{2i} \end{bmatrix} \cong (t_i) \begin{bmatrix} \sigma_{1i} \\ \sigma_{2i} \end{bmatrix}}{\begin{bmatrix} C t_{11} \dots t_{1n} \\ C t_{21} \dots t_{2n} \end{bmatrix} \cong (C t_1 \dots t_n) \begin{bmatrix} \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \\ \langle \sigma_{21}, \dots, \sigma_{2n} \rangle \end{bmatrix}} \\
 \\
 \text{LIST} \frac{\forall i \in [1, n], \begin{bmatrix} t_{1i} \\ t_{2i} \end{bmatrix} \cong (t_i) \begin{bmatrix} \sigma_{1i} \\ \sigma_{2i} \end{bmatrix}}{\begin{bmatrix} [t_{11}, \dots, t_{1n}] \\ [t_{21}, \dots, t_{2n}] \end{bmatrix} \cong ([t_1, \dots, t_n]) \begin{bmatrix} \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \\ \langle \sigma_{21}, \dots, \sigma_{2n} \rangle \end{bmatrix}} \\
 \\
 \text{OTHERWISE} \frac{}{\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (\text{Var } \alpha) \begin{bmatrix} (\alpha \mapsto t_1) \\ (\alpha \mapsto t_2) \end{bmatrix}}
 \end{array}$$

Figure 5.4: Inference rules to calculate the anti-unifier t for the terms t_1 and t_2 .

where p denotes any additional assumptions necessary for $t_1 \cong t\sigma_1$ and $t_2 \cong t\sigma_2$ to hold. We use matrix notation as shorthand for the two equations; i.e.

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (t) \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} = t_1 \cong t\sigma_1 \wedge t_2 \cong t\sigma_2$$

When t_1 and t_2 are the same, i.e. $t_1 \cong t_2$, EQ holds. For all other cases

when t_2 is a variable term, and is not $(\text{Var } p)$, t_2 is used as the hedge variable in t (VAR). When t_2 is an application term applying some function u to a v_{a_i} , and t_2 is a variable term that is equivalent to the v_{a_i} in t_2 , then the application term is used in t , and u is substituted for id in σ_1 (ID). When t_2 is an unobstructive recursive prefix of p , RP holds. CONST states that when t_1 and t_2 are terms with the same constructor, C , then t is a term with constructor C and its direct subterms are the anti-unifiers of each of the respective direct subterms in t_1 and t_2 . LIST is similar to CONST , but satisfies when the t_1 and t_2 are lists of terms, where t_1 and t_2 are the same length. In all other cases, OTHERWISE holds. For example, the terms, t_1 and t_2 ,

$$\begin{aligned} t_1 &= \text{App}(\text{Var } e) (\text{Lit } 42) \\ t_2 &= \text{App}(\text{Var } g) (\text{Lit } 108) \end{aligned}$$

both apply some function to some literal, and have the anti-unifier and substitutions:

$$\begin{aligned} t &= (\text{App}(\text{Var } g) (\text{Var } \alpha)) \\ \sigma_1 &= \langle (g \mapsto (\text{Var } e)), (\alpha \mapsto (\text{Lit } 42)) \rangle \\ \sigma_2 &= \langle \varepsilon, (\alpha \mapsto (\text{Lit } 108)) \rangle \end{aligned}$$

Here, σ_1 and σ_2 are compositions reflecting the two arguments to App (CONST). Since the first argument in both t_1 and t_2 are variable terms, but with different names, t_2 is chosen as the anti-unifier with the substitution for σ_1 replacing g with e , and the identity substitution for σ_2 (VAR). Conversely, the second argument differs in t_1 and t_2 , and since $(\text{Lit } 108)$ is not an identifier, a fresh hedge variable, α , is used for t , with the corresponding substitutions replacing $(\text{Var } \alpha)$ in $(\text{Lit } 42)$ and $(\text{Lit } 108)$ for t_1 and t_2 , respectively (OTHERWISE).

As we are interested in finding instances of a pattern in a function, it is convenient to relax the *least general* property found in traditional anti-unification algorithms. Specifically, the RP rule in Figure 5.4 can result in an anti-unifier that is not the least general generalisation of two terms. To illustrate this, consider the example of `elem` anti-unified against a `foldr`, where

```

1 elem a [] = False
2 elem a (x:xs) =
3   (\y ys -> if a == y then True else ys)
4   x (elem a xs)
5
6 foldr g z [] = z
7 foldr g z (x:xs) = g x (foldr g z xs)
    
```

Here, the anti-unification of the recursive calls in the *cons*-clauses produces the term:

$$\text{App}(\text{Var } \alpha)(\text{Var } xs)$$

since $(\text{foldr } g \ z)$ is an unobstructive recursive prefix. In this case, the RP rule applies instead of CONST. CONST would otherwise apply since both $(\text{foldr } g \ z)$ and $(\text{elem } a)$ are application expressions, and would produce a less general generalisation than the result of applying the RP rule. Intuitively, the least general generalisation is the term t that shares the closest structure to t_1 and t_2 , and requires the least number of substitutions in σ_1 and σ_2 . In the above example, the least general generalisation for the recursive call is:

$$\text{App}(\text{App}(\text{Var } \alpha)(\text{Var } z))(\text{Var } xs)$$

The ‘shortcuts’ produced by application of the RP rule are *useful*. Unobstructive recursive prefixes are uninteresting, aside from their relative location, since they produce no valid *argument candidates* in σ_{f} (Section 5.4). Moreover, without the RP rule, extra work would be necessary to derive and/or choose between argument candidates.

Theorem 5.1 (Soundness of Anti-Unification Algorithm). *Given the terms t_1 and t_2 , we can find t , σ_1 , and σ_2 , such that $t_1 \cong t \sigma_1$ and $t_2 \cong t \sigma_2$ hold.*

We give the proof for the above soundness property in Appendix B.

Having defined anti-unification for terms, we can now define it for \mathbb{f} and \mathbb{p} . Given the functions \mathbb{f} and \mathbb{p} ,

$$\begin{array}{ccc} \mathbb{f} \ v_{11} \ \dots \ v_{1m} = e_1^{\mathbb{f}} & & \mathbb{p} \ w_{11} \ \dots \ w_{1l} = e_1^{\mathbb{p}} \\ & \vdots & \vdots \\ \mathbb{f} \ v_{n1} \ \dots \ v_{nm} = e_n^{\mathbb{f}} & & \mathbb{p} \ w_{n1} \ \dots \ w_{nl} = e_n^{\mathbb{p}} \end{array}$$

where v_{ij} and w_{ij} are arguments to \mathbb{f} and \mathbb{p} , respectively, and we define the anti-unification of \mathbb{f} and \mathbb{p} , denoted \mathbb{h} , to be the list:

$$\mathbb{h} = [\llbracket e_1^{\mathbb{f}} \rrbracket \triangleq \llbracket e_1^{\mathbb{p}} \rrbracket, \dots, \llbracket e_n^{\mathbb{f}} \rrbracket \triangleq \llbracket e_n^{\mathbb{p}} \rrbracket]$$

where each element, $\forall i \in [1, n]$, $e_i^{\mathbb{h}}$, in \mathbb{h} corresponds to the anti-unification of the i^{th} clauses in \mathbb{f} and \mathbb{p} ; and elements are the triple $(t_i \times \sigma_{1i} \times \sigma_{2i})$. The substitutions for each element in \mathbb{h} can be composed, and we refer to the composition of substitutions for \mathbb{f} as $\sigma_{\mathbb{f}}$, and the composition of substitutions for \mathbb{p} as $\sigma_{\mathbb{p}}$; i.e.

$$\begin{aligned} \sigma_{\mathbb{f}} &= \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \\ \sigma_{\mathbb{p}} &= \langle \sigma_{21}, \dots, \sigma_{2n} \rangle \end{aligned}$$

\mathbb{h} is, in principle, equivalent to a function with n equations and where:
i) the anti-unifier term in each element represents the right-hand side of its respective equation; and *ii)* the set of all hedge variables are the parameters to the function. For example, the functions \mathbb{f} and \mathbb{p} ,

```
1  f x = x + 42
2  g y = y + 108
```

when anti-unified, produce:

$$\begin{aligned} \mathbb{h} &= [(\text{App} (\text{App} (\text{Var} +) (\text{Var } y)) (\text{Var } z)) \\ &\quad \times \langle \langle \varepsilon, (y \mapsto (\text{Var } x)) \rangle, (z \mapsto (\text{Id } 42)) \rangle \rangle \times \langle \langle \varepsilon, \varepsilon \rangle, (z \mapsto (\text{Id } 108)) \rangle \rangle] \end{aligned}$$

Here, \mathbb{h} can be represented as:

```
1  h y z = y + z
2
```

```

3  f x = h x 42
4
5  g y = h y 102

```

where, h is the anti-unifier, and the substitutions are represented using standard function application. To simplify our presentation, in examples we confuse h for its equivalent function definition.

5.4 Deriving Pattern Arguments

Given the anti-unifier h of f and p , with substitutions σ_f and σ_p respectively, we next derive the arguments to p that are needed to rewrite f as an instance of p . Our anti-unification algorithm is designed such that σ_1 provides *candidate* arguments. These candidates are considered *valid* when they adhere to three properties:

1. that p and h are equivalent (Definition 5.4.2);
2. that there does not exist a (sub-)substitution in σ_f or σ_p where a v_{a_i} occurs as either a hedge variable or as a term (Definition 5.4.4); and
3. that all substitutions in σ_f substituting for the same hedge variable that is derived from p must substitute for the same term (Definition 5.4.6).

Equivalence of Pattern and Anti-Unifier Recall that we aim to rewrite f in terms of p . As stated in Section 5.3, the syntactic equivalence properties of the produced anti-unifier allow f to be rewritten as a call to the function representation of h . Furthermore, by equational reasoning [101], if p and h are *equivalent*, then f can be rewritten to replace the call to h with a call to p ; i.e. f is an instance of p . Since anti-unification will *always* produce an anti-unifier and substitutions between any two arbitrary terms, the production of an anti-unifier *cannot* be used as a test of equivalence. We instead define an equivalence relation between p , h , σ_f , and σ_p . In order to do this, we first define equivalence between two terms t_2 and t and substitutions σ_1 and σ_2 with respect to t_1 .

Definition 5.4.1 (Equivalence of Terms with Substitutions). *Given the terms, t, t_1, t_2 , and the substitutions σ_1 and σ_2 , such that*

$$\begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \cong (t) \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}$$

we say that t_2 and t are equivalent with respect to σ_1, σ_2 , and t_1 , (denoted $t_2 \equiv_{t_1} t \times \sigma_1 \times \sigma_2$) when:

$$\text{EQ} \frac{t_2 \cong t}{t_2 \equiv_{t_1} t \times \varepsilon \times \varepsilon} \quad \text{HEDGE} \frac{v \neq p \quad v \in \mathcal{V}_p}{(\text{Var } v) \equiv_{t_1} (\text{Var } v) \times (v \mapsto t_1) \times \varepsilon}$$

$$\text{FRESH} \frac{t_1 \in \mathcal{R}_f \quad t_2 \in \mathcal{R}_p \quad \alpha \notin \mathcal{V}_p}{t_2 \equiv_{t_1} (\text{Var } \alpha) \times (\alpha \mapsto t_1) \times (\alpha \mapsto t_2)}$$

$$\text{CONST} \frac{\forall i \in [1, n], t_{2i} \equiv_{t_1} t_i \times \sigma_{1i} \times \sigma_{2i}}{(C \ t_{21} \ \dots \ t_{2n}) \equiv_{t_1} (C \ t_1 \ \dots \ t_n) \times \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \times \langle \sigma_{21}, \dots, \sigma_{2n} \rangle}$$

$$\text{LIST} \frac{\forall i \in [1, n], t_{2i} \equiv_{t_1} t_i \times \sigma_{1i} \times \sigma_{2i}}{[t_{21}, \dots, t_{2n}] \equiv_{t_1} [t_1, \dots, t_n] \times \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \times \langle \sigma_{21}, \dots, \sigma_{2n} \rangle}$$

Here, EQ states that when t_2 and t are the same, and when σ_1 and σ_2 are both ε , then t_2 and t are equivalent with respect to σ_1 and σ_2 . When t_2 and t are both variable terms, t_2 and t are equivalent when the variables have the same name and when the variable is a hedge variable derived from p (HEDGE). Similarly, when t_2 and t are both fresh hedge variables, α , then both terms substituted for α must be unobstructive recursive prefixes in their respective functions (FRESH). Finally, CONST and LIST state that terms consisting only of subterms are equivalent when all respective subterms are equivalent. The definition of equivalence for p and h then follows naturally.

Definition 5.4.2 (Equivalence of Pattern and Anti-Unifier). *Given two functions \mathfrak{f} and \mathfrak{p} , and the result of their anti-unification, \mathfrak{h} , containing the substitutions $\sigma_{\mathfrak{f}}$ and $\sigma_{\mathfrak{p}}$, we say that \mathfrak{p} and \mathfrak{h} are equivalent when for all clauses in \mathfrak{p} , $e_i^{\mathfrak{p}}$, and elements in \mathfrak{h} , $e_i^{\mathfrak{h}} = (t_i \times \sigma_{1i} \times \sigma_{2i})$, $\llbracket e_i^{\mathfrak{p}} \rrbracket \equiv_{t_{1i}} e_i^{\mathfrak{h}}$.*

Absence of Shared Arguments in Substitutions As stated in Section 5.2, the variables, v_{a_i} , that are declared in a_i are shared between f and p . As such, if f is an instance of p , v_{a_i} must be used in f in an equivalent way to their use in p ; i.e. their relative locations in the structure of f and p must be the same. For example, in `sumeuler` and `foldr`,

```

1  sumeuler [] = 0
2  sumeuler (x:xs) = ((+) . euler) x (sumeuler xs)
3
4  foldr g z [] = z
5  foldr g z (w:ws) = g w (foldr g z ws)

```

the heads of the matched lists, x in `sumeuler` and w in `foldr`, are both passed as the first argument to some function, g ; and the tails of the matched lists, xs and ws respectively, are passed as the last argument to the recursive calls in f and p . While the equivalence of pattern and anti-unifier (Definition 5.4.2) ensures the structural equivalence of p and h , it does not inspect hedge variables derived from p . Our anti-unification rules in Figure 5.4 means that no v_{a_i} will occur as either a hedge variable or as a subterm in either σ_f or σ_p when all equivalent v_{a_i} have the same relative positions in f and p . Conversely, the occurrence of some v_{a_i} in σ_f or σ_p indicates that not all equivalent v_{a_i} have the same relative positions in f and p .

Definition 5.4.3 (Variable Occurrence in Terms). *For all variables v , and for all terms, t , we say that v occurs in t (denoted $v \ll t$) when $t = (\text{Var } v)$ or when there exists a subterm, t_i , in t such that $t_i = (\text{Var } v)$.*

Definition 5.4.4 (Shared Argument Absence). *Given the functions f and p , and their anti-unification h , containing the substitutions σ_f and σ_p , we say that no v_{a_i} occurs in σ_f or σ_p (denoted $v_{a_i} \notin (\sigma_f, \sigma_p)$) when, for all v_{a_i} :*

$$\frac{v \not\ll v_{a_i} \quad (\text{Var } v_{a_i}) \not\ll t}{v_{a_i} \notin (v \mapsto t)} \quad \frac{\forall i \in [1, n], v_{a_i} \notin \sigma_i}{v_{a_i} \notin (\sigma_1, \dots, \sigma_n)}$$

Substitution Uniqueness Substitution composition (Definition 5.3.3) ensures that the same hedge variable can be used to substitute for different terms without the problem of substitution interference. For example, consider some f that is anti-unified against `scanl`:

```

1  f a [] = [a]
2  f a (x:xs) = (+) a x : f ((-) a x) xs
3
4  scanl g z [] = [z]
5  scanl g z (x:xs) = g z x : scanl g (g z x) xs

```

Here, the result of the anti-unification of the two *cons*-clauses will include substitutions where g is substituted for both $(+)$ and $(-)$ in σ_f . In `scanl`, however, g is the same in both instances. We therefore also require that all substitutions that substitute for the same hedge variable in σ_f are substituted for the same term.

Definition 5.4.5 (Substitution Flattening). *For all substitutions, σ , we say the flattening of σ (denoted $\lfloor \sigma \rfloor$) is the list of substitutions, where:*

$$\frac{}{\lfloor (v \mapsto t) \rfloor = [(v \mapsto t)]} \quad \frac{}{\lfloor (\sigma_1, \dots, \sigma_n) \rfloor = \lfloor \sigma_1 \rfloor + \dots + \lfloor \sigma_n \rfloor}$$

Here, $+$ appends two lists, and $[a]$ denotes the singleton list.

A list of substitutions cannot be applied to a term, allowing us to safely expose any potential substitution interference across f .

Definition 5.4.6 (Substitution Uniqueness). *Given the functions f and p , and their anti-unification h . Given that σ_f is the composition of substitutions in h for the clauses of f . We say that substitutions are unique in σ_f when for all hedge variables, $v \in \mathcal{V}_p$, and for all pairs of substitutions in $\lfloor \sigma_f \rfloor$, $(v \mapsto t_1)$ and $(v \mapsto t_2)$, $t_1 = t_2$ holds.*

Sufficiency of Validity Properties Given f , p , h , σ_f and σ_p , the above three properties are sufficient to determine whether the candidate arguments in σ_f are valid. Equivalence of p and h (Definition 5.4.2) ensures that for all clauses in p and h , the terms for that clause, t_2 and t , are syntactically equivalent (Definition 5.2.3). Hedge variable terms have

additional information in their respective substitutions, σ_1 and σ_2 . There are two cases: fresh hedge variables, α , and hedge variables derived from p , v . For α , Definition 5.4.2 requires that both terms, t_1 and t_2 , substituted for α are unobstructive recursive prefixes (Definition 5.3.4). No other terms are allowed. For v , two cases are possible: $v = v_{a_i}$ and otherwise. All equivalent v_{a_i} in f and p must have the same relative locations to ensure that a_i is traversed in the same way, and that no free variables occur in any argument candidate expression. The rules in Figure 5.4 mean that the identity substitution is derived for all equivalent v_{a_i} in the same relative position in f and p . For the case when a v_{a_i} is not anti-unified with its equivalent, Shared Argument Absence (Definition 5.4.4) requires that no v_{a_i} occurs in any substitution and so will fail in this case. For all other v , Definition 5.4.2 requires that v will be in scope, due to the assumption that all variables that occur in p are in scope and are declared as parameters of p . Any updates to the value of v must be reflected in p , e.g. such as in `foldl` and `scanl`. Finally, and since each argument may be instantiated only once for each call to p , Substitution Uniqueness (Definition 5.4.6) ensures that for all substitutions that substitute for the same hedge variable, the substituted terms are the same.

5.4.1 Deriving Arguments from Substitutions

Given that f , p , and their anti-unification, h , adhere to Definition 5.4.2 (Equivalence of Pattern and Anti-Unifier), Definition 5.4.4 (Shared Argument Absence), and Definition 5.4.6 (Substitution Uniqueness), the arguments for p can be directly obtained from σ_f . For `sumeuler`, given σ_1 for the `[]` clause:

$$\sigma_1 = (z \mapsto (\text{Lit } 0))$$

and σ_1 for the `(:)` clause:

$$\sigma_1 = \langle \langle (g \mapsto (\text{App} (\text{App} (\text{Var } .) (\text{Var } +)) (\text{Var euler}))), \epsilon \rangle, \langle (\alpha \mapsto (\text{Var sumeuler})), \epsilon \rangle \rangle$$

`((+) . euler)` is passed as g ; `0` is passed as z ; and a_i (i.e. `(x:xs)`) is passed as itself due to Shared Argument Absence and Argument

Equivalence properties. Substitutions with fresh hedge variables are discarded. We can now refactor f in terms of p using the derived arguments above, to give:

```
1 sumeuler xs = foldr ((+) . euler) 0 xs
```

Which, as before, can be rewritten to give our original definition:

```
1 sumeuler xs = foldr (+) 0 (map euler xs)
```

5.5 Implementation

We have implemented our approach as an extension to the Haskell refactoring tool, HaRe. In this section we give an overview of our implementation, highlighting differences from the description of our approach.

Our prototype extends version 0.6 of HaRe [21], which requires GHC version 6.12.1, and it was developed in a Debian 6.0.10 virtual machine using Virtual Box 5.0.32. While newer versions of HaRe are available, 0.6 is the latest version prior to the rewrite which saw the selection of available refactorings reduce from 53 refactorings to six refactorings. This was to use GHC's internal AST representation, which was not available at the time of Brown's thesis. Brown's clone detection and elimination implementation in HaRe [23], which uses anti-unification to detect clones, was also removed in versions 0.7 onwards. We chose to extend version 0.6 because of the wider selection of existing refactorings, which can potentially be used during normalisation, and although ultimately unused, because we considered using Brown's anti-unification algorithm. It should be possible to update our prototype implementation for use with later versions of HaRe, but we defer this to future work.

At the core of our prototype is the top-level function, `auFPair`, which serves as an implementation of our approach. Our prototype then features a wrapper refactoring, *Anti-Unify Module* that enables the programmer to invoke pattern discovery, given a pattern module, i.e. a Haskell module of declarations to be used as p . Additional refactorings with default pattern modules are provided for `map`, `fold`, `zipWith`, and `scanl` recursion schemes.

Our implementation reflects the anti-unification algorithm in Section 5.3, verification checks in Section 5.4, argument inference and rewriting as described in Section 5.4.1, and includes some simple normalisations. As implemented, normalisation validates the format of arguments, i.e. all but the last argument patterns matched are variables, and variables, a_i , consists of a single-depth pattern match. Infix applications, infix patterns, and left-section expressions are rewritten to equivalent prefix expressions or patterns. f and p are checked for having the correct number of clauses, and the clauses are in the same relative order according to matched constructor. Finally, v_{a_i} are renamed to their respective de Bruijn indices in both f and p . For example, `isums`, is normalised to:

```

1  isums b [] = []
2  isums b ((:) h t) =
3    ((:) (((+) h) b)) ((isums (((+) h) b)) t)

```

Note that additional brackets have been introduced to highlight application (sub)expressions. Here, our implemented normalisation stage checks that b , h , and t are variables, and that the list constructors are the last passed argument to `isums`. The infix applications of `(+)` and `(:)` have been transformed into prefix forms. The normalisation stage will also check that both `[]` and `(:)` constructors are matched in exactly one clause, wherever p is some pattern in \mathcal{P} defined over a single list.

We make an additional assumption for convenience of implementation when τ is a product type. All arguments that are recursed over must be tupled. For example, both the traversed lists in the standard `zipWith` definition from the Haskell prelude:

```

1  zipWith g [] [] = []
2  zipWith g (x:xs) (y:ys) = g x y : zipWith g xs ys

```

can be tupled, perhaps using the *Curry/Uncurry Arguments* refactoring in HaRe, such that a_i is a single argument:

```

1  zipWith g ([], []) = []
2  zipWith g ((x:xs), (y:ys)) = g x y : zipWith g (xs, ys)

```


5.6 Finding unfold

Thus far, we have looked exclusively at schemes which can be expressed in terms of `fold`. Schemes with *other* structures may require different handling. In this section, we consider `unfold` [47]. The *unfold corecursively* generates a data structure, τ , from a seed value. An `unfold` that generates a cons-list, for example, can be defined:

```
1 unfold :: (a -> Either [b] (b,a)) -> a -> [b]
2 unfold g x = case g x of
3             Left  zs -> zs
4             Right (y,x') -> y : unfold g x'
```

where `g` is a function that takes a seed value, `x`, and generates the components necessary for the list to be constructed. The structure of the `unfold` then arranges these components into constructors of τ . As with its categorical dual, `fold`, the `unfold` family of recursion schemes can be generated for arbitrary data types [47].

Consider a simple *quicksort* example, *quicksort* can be defined as a composition of `join` and `split`, *qs*:

```
1 qs xs = join (split xs)
2   where
3     join Empty = []
4     join (Node x l r) = l ++ (x : r)
5
6     split []      = Empty
7     split (x:xs) = Node x
8                   (split (filter (<= x) xs))
9                   (split (filter (> x) xs))
```

Here, `split` takes a *cons*-list as its argument, and generates a tree. `split` *cannot* be defined as a `fold` as the tail of the input list is changed with each recursive call (which folds do not allow). Instead of viewing `split` as a `fold` over a cons-list, we can instead view it as an `unfold` creating a binary tree, redefining `split`,

```

1 split :: Ord a => [a] -> Tree a
2 split xs = unfold g xs
3   where
4     g [] = Nothing
5     g (x:xs) =
6       Just (x, (filter (<= x) xs), (filter (> x) xs))

```

Our approach will not discover the unfold in this split definition for two reasons:

1. split and unfold will not be deemed equivalent (Definition 5.4.2), and
2. our discovery approach does not inspect functions that are called in the body of f or p.

In order to solve the first problem, we refactor the original definition of split to introduce an intermediate function, g' :

```

1 split :: Ord a => [a] -> Tree a
2 split xs =
3   case g' xs of
4     Nothing -> Empty
5     Just (x, lxs, rxs) ->
6       Node x (split lxs) (split rxs)
7   where
8     g' [] = Nothing
9     g' (x:xs) =
10      Just (x, (filter (<= x) xs), (filter (> x) xs))

```

Here, we have lifted the pattern match over the input list into a function which returns a Maybe constructor corresponding to the constructor that was originally produced. Here, Nothing is mapped to Empty (since Empty has no arguments) and Just is mapped to Node. We use Maybe for convenience and familiarity for descriptive purposes; we could alternatively define, and mechanically derive, a data type with constructors, C1 and C2 that serve the same purpose. Introduction of g' separates the generation of the parts needed to ultimately construct the tree, from

the mechanics of corecursive generation. Similarly, we conjecture that g' can be generated automatically using equational reasoning to match constructors and arguments.

Despite the relative structures of `split` and `unfold` now being equivalent, our approach still cannot discover the arguments for `unfold` to rewrite `split`. Our approach inspects individual pairs of functions, and does not inspect functions called within either `f` or `p`. This affects `split` and `unfold` because Section 5.2 requires that all `case`-expressions be lifted out as functions. Following this requirement, `split` and `unfold` are redefined as:

```

1  unfold :: (a -> Maybe (b,a,a)) -> a -> Tree b
2  unfold g x = unfold' g (g x)
3  where
4    unfold' g Nothing = Empty
5    unfold' g (Just (z,x',x'')) =
6      Node z (unfold g x') (unfold g x'')
7
8  split :: Ord a => [a] -> Tree a
9  split xs = split' (g' xs)
10 where
11   g' [] = Nothing
12   g' (x:xs) =
13     Just (x, (filter (<= x) xs), (filter (> x) xs))
14
15   split' Nothing = Empty
16   split' (Just (x, lxs, rxs)) =
17     Node x (split lxs) (split rxs)

```

After this change, our HOF discovery approach will inspect only the single clauses of `unfold` and `split` but not the auxiliary functions `split'` and `unfold'`. Since all introduced hedge variables are assumed to substitute for recursive prefixes, `split` will never be rewritten in terms of `unfold` because `unfold'` is not a variable passed to `unfold`. In order to discover `unfold`, we modify our approach to consider and inspect *mutually recursive* definitions.

5.6.1 Assumptions

All the assumptions from Section 5.2 still hold. Additionally, we assume that an `unfold` consists of a single clause that calls an auxiliary function. The `unfold` and its auxiliary function are *mutually recursive*. The common ground, a_i , in an `unfold` definition will always be a single variable, v_{a_i} . The auxiliary function is passed all arguments, apart from v_{a_i} , which is passed as an argument to some fixpoint function, g . For example

```

1  unfold :: (a -> Maybe (b, a, a)) -> a -> Tree b
2  unfold g x = unfold' g (g x)
3  where
4    unfold' g Nothing = Empty
5    unfold' g (Just (z, x', x'')) =
6      Node z (unfold g x') (unfold g x'')
```

Intuitively, we extract a function that pattern matches over a type, whose clauses we can then compare. f should be normalised to this form, perhaps by applying a series of refactorings.

To simplify our proof of concept implementation of `unfold`-discovery, we denote the auxiliary function of f , to be f' , and assume that f' is defined in the `where`-block of f .

5.6.2 Method and Implementation

Both f and `unfold` are first normalised, and then anti-unified as before. The definition of a recursive prefix is extended to include calls to mutually recursive functions. For example, `split` and `unfold` above are anti-unified, producing:

```
1  h g x =  $\alpha$  (g x)
```

with substitutions:

$$\begin{aligned}\sigma_1^1 &= \llbracket (\alpha \mapsto \text{split}', (g \mapsto g')) \rrbracket \\ \sigma_2^1 &= (\alpha \mapsto \text{unfold}' \ g)\end{aligned}$$

Here the fresh hedge variable, α , is introduced to account for the unobstructive recursive prefixes `(unfold' g)` and `split'`.

Assuming that the requirements described in Section 5.4 hold for f , unfold , h , σ_1 , and σ_2 , then the auxiliary function declarations, f' and p' , can be found, normalised, and anti-unified. Anti-unification of split' and unfold' produces:

```

1 h' g Nothing = Empty
2 h' g (Just (z, x', x'')) = Node z (α1 x') (α2 x'')
```

with the substitutions:

$$\begin{aligned}
\sigma_1^1 &= \varepsilon \\
\sigma_1^2 &= \langle \langle (z \mapsto x), (\alpha_1 \mapsto \text{split}'), (x' \mapsto \text{lhs}) \rangle \rangle, \\
&\quad \langle \langle \alpha_2 \mapsto \text{split}', (x'' \mapsto \text{rhs}) \rangle \rangle \\
\sigma_2^1 &= \varepsilon \\
\sigma_2^2 &= \langle \langle \alpha_1 \mapsto \text{unfold}' \ g, (\alpha_2 \mapsto \text{unfold}' \ g) \rangle \rangle
\end{aligned}$$

As before, assuming that the requirements described in Section 5.4 hold for f' and p' , we extend the *substitution uniqueness* requirement (Definition 5.4.6) across both sets of substitutions. unfold arguments for f are then inferred. Substitutions for f' and p' are ignored, since arguments to unfold cannot affect unfold' . In addition to the assurances described in Section 5.4, and the extension of duplicate substitutions, the assumption that arguments to p' consist only of arguments passed to unfold and the result of the application of g to v_{a_i} , mean that no additional arguments should be inferred from auxiliary substitutions. Returning to the split example, in the auxiliary function substitutions, all hedge variables are either fresh, or are v_{a_i} -equivalent functions and are ignored. Conversely, in the set of substitutions for split and unfold , we infer g . This enables us to rewrite split as:

```

1 split x = unfold g' x
2   where
3   g' [] = Nothing
4   g' (x:xs) =
5     Just (x, (filter (<= x) xs), (filter (> x) xs))
```

5.7 Examples

In this section, we demonstrate our approach using the five parallel benchmarks described in Section 4.6.2, a collection of examples from the Haskell Prelude, and examples derived from Cook’s and Kannan’s theses [41, 74]. Each example function is applied to our prototype, which we describe in Section 5.5. In our examples, we sought to discover instances of `map`, `foldr`, `foldl`, `scanl`, and `zipWith` schemes. We give a summary of the number of scheme instances found in Figure 5.5, including the average time taken by our prototype (and standard deviations) in seconds. In Section 5.7.6, we compare our anti-unification approach with our slicing approach to pattern discovery, introduced in Chapter 4.

5.7.1 Normalisations Enacted

The source code of all examples has been normalised to facilitate the use of our prototype, where normalisation processes include: flipping argument ordering, removing parallel constructs and duplicate sequential code, and inlining existing schemes. We normalise the sequential version of each example when it is present, and the parallel version that gives best speedups when a sequential version is not available. Normalisations include:

- We have inlined explicit calls to `map`, `foldr`, `foldl`, `scanl`, and `zipWith`.
- We do not inline specialised patterns, e.g. `sum`, in parallel; instead, these are demonstrated in the `Data.List` set of examples.
- We have removed parallel constructs, duplicate definitions, and monadic code.
- We have removed `main` definitions.
- We have rewritten calls to `repeat` to `replicate`, in order to ensure correct pattern matching when passed to, e.g., `zipWith` pattern instances.

Example	Funs	Map	Fold	Scan	Zip	None	Time	SD
Data.List	34	4	17	1	2	10	1.55	0.02
Cook	25	5	16	1	0	3	1.33	0.04
Kannan	22	1	8	1	0	12	1.04	0.02
ListAux	12	1	2	0	2	7	0.74	0.01
Queens	7	1	2	0	0	4	0.62	0.01
MatMult	6	2	1	0	0	3	0.59	0.02
nbody	7	1	1	0	0	5	0.71	0.05
sudoku1	1	1	0	0	0	0	0.13	0.00
Sudoku	38	10	7	0	5	16	1.75	0.04
SumEuler	7	0	2	0	0	5	0.27	0.01
SumEulerPrimes	11	1	3	0	0	7	0.81	0.04

Figure 5.5: Examples run through prototype implementation. Times are given in seconds and are an average of five runs. Unfold tests are separate.

- We have tupled arguments where a function traverses multiple data structures.

5.7.2 Experimental Setup

Since our prototype requires GHC 6.12.1 and has been developed using a virtual machine, *ananke*, running Debian 6, all reported times are the average of five runs on this virtual machine. *ananke* was created using Parallels Desktop version 11.2.2 with performance settings set to favour the virtual machine and the Adaptive Hypervisor enabled. It has been allocated two 3.4GHz processors, and 1GB of memory. *ananke* is running on a standard desktop machine, *jupiter*, a 3.4GHz Intel Core i7 machine with 4 physical cores (8 with hyperthreading enabled) and 32GB of RAM, running macOS 10.12.5.

5.7.3 Parallel Benchmarks

We demonstrate our approach using the five parallel benchmarks presented in Section 5.7.3: `sudoku`, `sumeuler`, `queens`, `nbody`, and `matmult`. We have applied our prototype to the normalised code in each example, and where multiple implementations of the same example are present,

we applied our prototype to the sequential version, if present, or the normalised functions that provide best speedups. For the below examples, our anti-unification approach is able to discover the same `map` instances as our slicing approach in Chapter 4 when the same function has equivalent normalisations for both slicing and anti-unification approaches. Alternative approaches may yield different results. Such results could also potentially disagree; e.g. if a `map` instance is normalised such that our slicing approach discovers the `map` but our anti-unification approach discovers a `foldr`.

Sudoku

The `sudoku` benchmark consists of nine Haskell files that comprise a sequential implementation and various parallelisations. The sequential implementation comprises the module `Sudoku` (150 lines) that contains functions to solve individual puzzles, and the file `sudoku1.hs` (16 lines) that contains one function that reads in a list of puzzles and solves them by applying `solve`. The normalised implementation of `sudoku1.hs` comprises a single function, i.e.

```
1 sudoku [] = []
2 sudoku (p:ps) = solve p : sudoku ps
```

that we know to be a `map` instance. The normalised implementation of `Sudoku` comprises a total of 38 functions, of which there are ten `map` instances, seven `fold` instances, no `scan` instances, five `zipWith` instances, and 16 non-recursive functions that are not an instance of a pattern in \mathcal{P} .

Sumeuler

The `sumeuler` benchmark consists of three Haskell files: `ListAux` (41 lines), `SumEulerPrimes` (36 lines), and `SumEuler` (290 lines). The normalised version of `ListAux` comprises 12 functions, of which there are two `fold` instances, two `zipWith` instances, a `map` instance, no `scan` instances, and seven non-recursive functions that are not pattern

instances. The normalised `SumEuler` comprises seven functions, of which two are `fold` instances,

```
1 sumEuler [] = 0
2 sumEuler (x:xs) = ((+) . euler) x (sumEuler xs)
3
4 euler1 n [] = []
5 euler1 n (x:xs) = (\x z -> if relprime n x
6                      then x:z
7                      else z) x (euler1 n xs)
```

The remaining five functions are all non-recursive and therefore not pattern instances. The normalised `SumEulerPrimes` comprises 11 functions, of which three functions are `fold` instances, one function is a `map` instance, and seven functions are not pattern instances. Six of these seven functions are non-recursive. The remaining function, `primesIn`, is an interesting case:

```
1 primesIn :: [Int] -> Int -> [Int]
2 primesIn ps@(p:rest) n
3     | p > n = []
4     | n `mod` p == 0 = p:primesIn ps (n `div` p)
5     | otherwise = primesIn rest n
```

`primesIn` traverses an *infinite* list. Whilst it can be normalised such that it matches the second clause of `foldr`, since no behaviour is defined for the empty-list case, however, `primesIn` will not be rewritten.

N-Queens

The `queens` benchmark consists of a single module, `Main` (42 lines), that we will rename to `Queens` for the sake of clarity. The normalised `Queens` is based on the sequential version in the Imaginary set of the `NoFib` suite, and comprises seven functions, of which two are `fold` instances, one function is a `map` instance, and four functions are not pattern instances. Three of these four functions are not recursive; the remaining function, `gen`,

```

1 gen nq 0 = [[]]
2 gen nq n = gen1 nq (gen (n-1))

```

could be rewritten such that `n` is a `Nat`:

```

1 gen nq Z = [[]]
2 gen nq (S n) = gen1 nq (gen n)

```

and could then be successfully rewritten as a `foldr` over `Nat` by our prototype:

```

1 gen nq n = foldrNat (gen nq) [[]] n
2
3 foldrNat g z Z = []
4 foldrNat g z (S n) = g (foldrNat g z n)

```

Another interesting function in `Queens` is `safe`; originally defined:

```

1 safe x d [] = True
2 safe x d (q:l) =
3     (x /= q)
4     && (x /= q+d)
5     && (x /= q-d)
6     && (safe x (d+1) l)

```

`safe` is considered a *near fold*. Since `d` is updated between recursive calls, our prototype implementation will not rewrite `safe` as a `foldl`. This is a correct result since a `foldl` that traverses the list will produce different results to the original definition. In order to transform this *near fold* into a `foldl`, we can move `d` into the initial value, by tupling `True` and `d`. `d` can then be updated via the fixpoint function. Finally, the value of `d` is discarded in favour of the boolean result.

```

1 safe x d qs = snd $ safel x (d, True) qs
2
3 safel x z [] = z
4 safel x z (q:qs) =
5   safel x ((\ (d, z) q ->
6             (d+1,
7              (x /= q) && (x /= q+d)
8              && (x /= q-d) && z)) z q) qs

```

The introduced `safel` can then be rewritten as a `foldl` instance by our prototype.

```

1 safel x z qs =
2   foldl (\ (d, z) q ->
3         (d+1, (x /= q) && (x /= q+d)
4         && (x /= q-d) && z) z qs

```

Since implicit `fold` instances may be *near fold* instances similar to `safe`, it may therefore be worth investigating this as a pattern or a normalisation stage.

N-Body

The `nbody` benchmark consists of two Haskell files: `Future` (16 lines) and `nbody.hs` (135 lines). `Future` comprises only custom implementations of strategies, and so we do not consider this module. Conversely, the normalised version of `nbody` comprises a total of seven functions, of which one is a `map` instance, one is a `fold` instance, and five functions are non-recursive and so are not an instance of any pattern. Both `map` and `foldr` instances are straightforward.

Matrix Multiplication

The `matmult` benchmark consists of two modules: `ListAux` (41 lines) and `MatMult` (246 lines). Both `sumeuler` and `matmult` share `ListAux`, and so we use the same normalisation here. `MatMult` features a lot of duplication, due to multiple parallelisations of a sequential implementation of matrix multiplication. The normalised sequential implementation

`in MatMult` comprises six functions, of which two are `map` instances, one is an instance of `foldl`, and three are non-recursive function definitions. Where both `map` implementations are straightforward, the implicit `foldl`, `addProd`, is strict in its accumulative argument (using the `BangPatterns` GHC extension).

```
1 addProd (v:vs) (w:ws) !acc =
2   addProd vs ws (acc + v*w)
3 addProd _ _ !n = n
```

Since `addProd` is a `foldl` over two lists, we tuple the two lists, make the accumulator, `acc`, the first argument to `addProd`, and lift the infix operators into a function. Since the second clause uses the wildcard pattern to return `n` for all combinations of patterns other than when both lists are *cons*, we are able to expand this clause into all remaining combinations of patterns.

```
1 addProd acc ([], []) = acc
2 addProd acc ((v:vs), []) = acc
3 addProd acc ([], (w:ws)) = acc
4 addProd acc ((v:vs), (w:ws)) =
5   addProd ((\z x y -> z + x * y) acc v w) (vs, ws)
```

In the resulting normalisation above the accumulator argument is no longer strict, which may have an effect on performance. One alternative is to maintain strictness via a higher-order function that supports it, e.g. `foldl'`.

5.7.4 Other Examples

Our other examples include 26 functions from the Haskell Prelude `Data.List` library, and two sets of examples from Cook's and Kannan's theses [41, 74]. All of Cook's examples are used as examples, but only a representative subset of Kannan's examples are used. Kannan's examples that are not used, include examples which are similar to examples we have used, or are similar to Cook's examples. As before, these examples may not all benefit from parallelisation, but

serve as a demonstration of our approach. All our normalised example code can be found at <http://adb23.host.cs.st-andrews.ac.uk/thesis-au-examples.zip>.

The 26 functions from the Haskell Prelude are a representative subset of the functions in the library; the remaining functions in `Data.List` are similar to those below. The functions include: `and`, `append`, `foldl`, `foldr`, `init`, `intersperse`, `last`, `length`, `map`, `maximum`, `or`, `scanl`, `replicate`, `reverse`, `subsequences`, `transpose`, `heads`, `tails`, `prependToAll`, `sum`, `splitAt`, `elem`, `filter`, `zip`, `zipwith`, `unzip`. Normalisation produces a total of 34 functions. Of these, 17 are `fold` instances, four are `map` instances, two are `zip` instances, one is a `scan` instance, and ten functions are not instances of any pattern in \mathcal{P} . Of the *AB: n* examples in `Data.List` that are also explored in Section 4.6.3, our anti-unification approach is able to discover the same `map` instances as our slicing approach in Chapter 4 when the function has equivalent normalisations. Of the ten functions that are not pattern instances, seven are non-recursive functions produced as part of the normalisation process to transform *near fold* functions into proper `fold` instances. Two interesting cases remain: *i*) `init` and its helper function, `init'`, and *ii*) `transpose`.

```
1  init (x:xs) =  init' x xs
2
3  init' _ []    = []
4  init' y (z:zs) = y : init' z zs
```

Here, `init` is a non-recursive function that passes the head and tail of its input list separately to `init'`. The helper function, `init'`, prepends the previous element, `y`, in the list to the result of the recursive call. In the recursive call, the current element, `z`, is passed to update `y`, and `zs` is passed as the second argument. Since `z` is passed as an argument to the recursive call, and that argument is not treated as an accumulator argument, as in e.g. `foldl`, our prototype concludes that `init'` is neither a `foldr` nor a `foldl` instance.

In the `Data.List` library, `transpose` is defined:

```

1 transpose [] = []
2 transpose ([] : xss) = transpose xss
3 transpose ((x:xs) : xss) =
4   (x : (heads xss)) : transpose (xs : (tails xss))

```

Here, `transpose` is not an example of *structural recursion* [68] since `xss` is not passed to the recursive call on Line 4. This is instead an example of *strong induction* [37], since the arguments to the recursive calls in Lines 2 and 4 are strictly smaller than the input list. The definitions of all patterns in \mathcal{P} are *structurally recursive*, and so no patterns are discovered. Given a pattern definition that is *strongly inductive*, and that when anti-unified with some valid normalisation of `transpose` produces an anti-unifier and substitutions that satisfy the conditions in Section 5.4, our prototype would be able to rewrite `transpose` as an instance of that pattern.

The second set of examples are derived from Cook’s thesis [41]. These examples include: calculating the squares of a two-dimensional list of numbers, calculating the sum of a two-dimensional list of numbers, `isums`, calculating the subset of a list, matrix multiplication, merge sort, and reversing a list. Contrary to other examples, we have normalised each function multiple times to maximise the number of patterns found. The result of this normalisation comprises 25 functions, of which 16 are `fold` instances, five are `map` instances, one function is a `scan` instance, and three functions are not pattern instances. As before, all three functions that are not pattern instances, are non-recursive functions. Our prototype is able to find all the patterns that Cook found using his approach.

The third and final set of examples are similarly derived from Kannan’s thesis [74]. We have normalised five examples, a representative subset of Kannan’s examples, and applied our prototype to these normalisations. The result of the normalisation process is 22 functions, of which there are eight `fold` instances, one `map` instance, one `scan` instance, and 12 functions that are not pattern instances. In contrast to Cook’s examples, Kannan’s examples feature functions that are *combinations* of patterns. For example, the maximum prefix sum, defined:

```

1 mps1 [] = 0
2 mps1 (x:xs) = mps2 xs x

```

```
3
4 mps2 [] v = v
5 mps2 (x:xs) v =
6   max v (max (x+v) (mps2 xs (x+v)))
```

Here, `mps2` is a combination of a `scanl` and a `foldr`; e.g.

```
1 mps2 v xs = mps2Fold (mps2Scan v xs)
2
3 mps2Scan v [] = [v]
4 mps2Scan v (x:xs) = (x+v) : mps2Scan (x+v) xs
5
6 mps2Fold v [] = v
7 mps2Fold v (x:xs) = f x (mps2Fold v xs)
8   where f x zs = max v (max x zs)
```

Here, `mps2Scan`, produces a list of prefix sums that are then folded over by `mp2Fold` that picks the element that represents the maximum. Our prototype was unable to discover the separate `scanl` and `foldr` instances of `mps2Scan` and `mps2Fold`, respectively. It is instead reliant on either the normalisation process splitting the original definition of `mps2`, or the provision of a pattern that represents the combination of a `scanl` and a `foldl` such that an anti-unifier and substitutions produced pass the requirements in Section 5.4.

For each of the above examples and parallel benchmarks, we have timed our prototype. Times for each example are presented in Figure 5.5, and are an average of five runs on *ananke*. Times have been derived using the Debian `time` package (version 1.7-24), taking the elapsed, wall-clock value. Using the *Anti-Unify Module* refactoring, all patterns in Appendix C have been applied to all functions in the respective example modules, providing an upper bound of analysis time. We found that all examples took less than two seconds to analyse. `Sudoku` took the longest time to analyse at 1.75s, with a standard deviation, σ , of 0.04s. Conversely, `sudoku1.hs` took the shortest time to analyse at 0.13s ($\sigma = 0.0045s$). Parallel and sequential runtimes for these parallel benchmarks are the same as those given in Section 4.6.2.

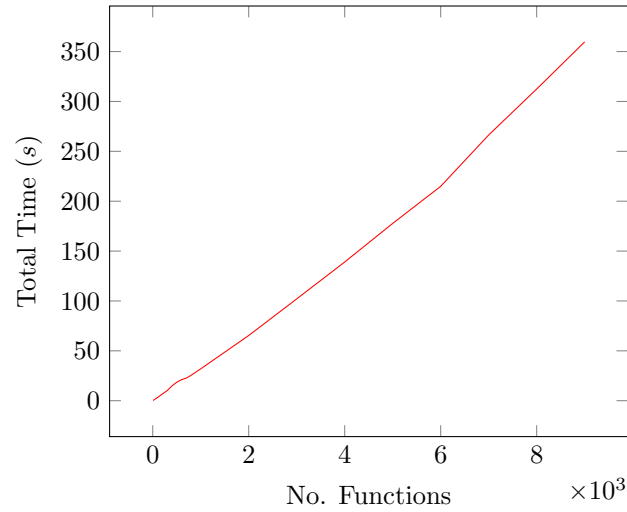


Figure 5.6: Prototype execution time for a program with varying sizes of input. This suggests that our implementation runs in linear time. Includes runs with initial module analysis operations automatically run by HaRe upon loading a fresh module. As a result, average runtimes are higher than in Figure 5.7.

As a synthetic benchmark, we have also applied our prototype to programs with varying input sizes, measured in the number of functions, by duplicating `append` n times. Figure 5.6 shows the average time taken of five runs, in seconds, by our prototype on *ananke*. It takes a minimum of 0.26s ($\sigma = 0.0084s$) with 5 functions, and a maximum of 359.70s ($\sigma = 178.69s$) for 9,000 functions. Standard error and distributions increase dramatically for inputs of greater than 1,000 functions. This is a result of HaRe parsing and analysing new files before it applies the refactoring. This analysis occurs every first run; Figure 5.7 shows the average time taken by our prototype with the first run removed. Here, we observe a minimum of 0.26s ($\sigma = 0.0084s$) with 5 functions, and a maximum of 279.79s ($\sigma = 1.73s$) for 9,000 functions. *ananke* runs out of memory for 10,000 functions and above. Times and memory footprint are likely to have been inflated by the output to `stdout`; all print statements were left in to ensure all examples are evaluated to normal form. Our prototype runs in linear time, with respect to the number of compared functions.

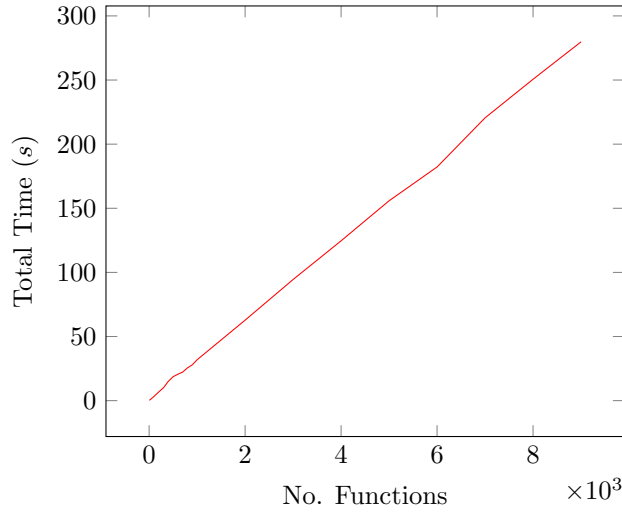


Figure 5.7: Prototype execution time for a program with varying sizes of input. Initial runs removed. This suggests that our implementation runs in linear time.

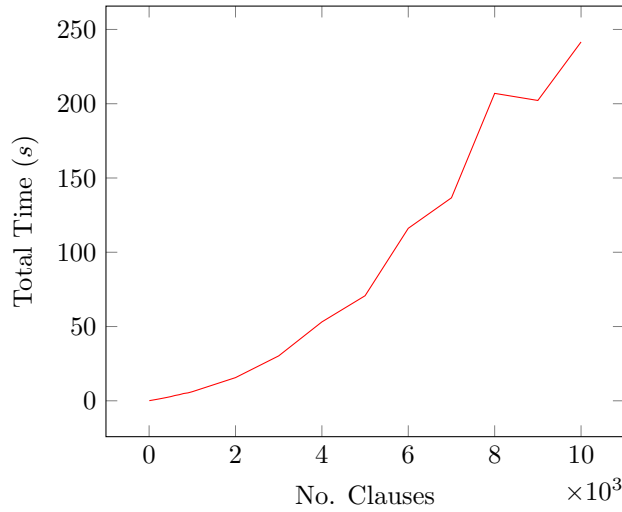


Figure 5.8: Prototype execution time for a program with varying sizes of input, measured in number of clauses. This suggests that our implementation runs in exponential time.

As a second synthetic benchmark, we have applied our prototype to two functions with varying numbers of clauses, by duplicating the second clauses of `append` and `foldr`, respectively. Figures 5.8 and 5.9 show the average time taken of five runs, in seconds, by our prototype on *ananke*. Figure 5.9 is the result of the first run removed. Once again, the analysis

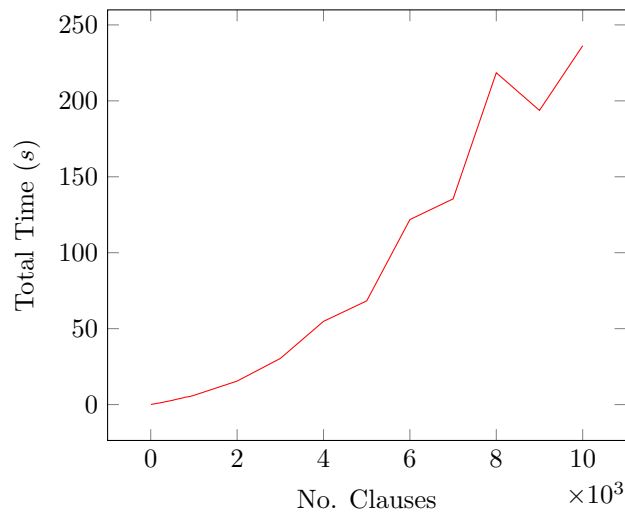


Figure 5.9: Prototype execution time for a program with varying sizes of input, measured in number of clauses. Initial runs removed. This suggests that our implementation runs in exponential time.

in each first run significantly increases variation in runtimes. We get an average of 0.10s ($\sigma = 0.00s$) for a six-clause function, and an average of 236.26s ($\sigma = 3.08s$) for a 10,001 clause function (with first runs removed). This suggests that our algorithm runs in exponential time, with respect to the number of clauses in both compared functions. Whilst our algorithm appears to run in exponential time, the exponential coefficient appears to be very small (7.30×10^{-4}), and has an asymptote at approximately 18,000 clauses.

5.7.5 Finding `unfold` Patterns

All examples presented in the above sections have focussed on the discovery of patterns that can be represented as instances of a `fold`. In Section 5.6, we proposed an extension to our approach to allow the discovery of other structures, using the example of the `unfold` pattern. Our extension allowed the discovery of a specific `unfold` pattern:

```

1  unfold f x = unfold' f (f x)
2  where
3    unfold' f c1 = []
4    unfold' f (c2 y ys) = y : (unfold f ys)
```

We have tested this pattern on three relatively simple examples: *i*) `unfold` itself, *ii*) a function that produces a co-recursive list of square numbers, and *iii*) `replicate` from the Haskell Prelude. Applying our extended approach to `unfold` successfully rewrites `unfold` as a call to the `unfold` definition used as a pattern. We use the following definition to co-recursively produce a list of square numbers,

```
1 sqs n x = sqs' n (sq n x)
2   where
3     sq n x  x < n = C2 (x * x) (x+1)
4              otherwise = C1
5
6     sqs' n C1 = []
7     sqs' n (C2 y z) = y : sqs n z
```

Our prototype successfully rewrites `sqs` as a call to `unfold` where `f` is `sqs'`, and `x` is `x`. The final example is taken from the Haskell Prelude. While we normalised `replicate` to

```
1 replicate n x = replicate1 x [1..5]
2
3 replicate1 y [] = []
4 replicate1 y (x:xs) = const y x : (replicate1 y xs)
```

in `Data.List`, we might alternatively rewrite it as the co-recursive definition:

```
1 replicate a n = replicate' a (a n)
2   where
3     replicate' a Z = []
4     replicate' a (S m) = a : replicate a m
```

Here, `n` has been converted to a natural number representation. Unlike `sqs`, `replicate` and `replicate'` do not apply any intermediate function (i.e. `f` in `unfold` or `sq` in `sqs`). Our prototype will therefore not rewrite `replicate` as an instance of `unfold`. To solve this, we might introduce a function that translates the natural number to the type that has `C1` and `C2` as constructors. This could be derived by equational

reasoning, by comparing pattern matches over `n` and `x`, and using the substitutions from anti-unifying the right-hand sides of the equations of `replicate'` and `unfold'`.

5.7.6 Comparison with Slicing Approach

The approaches introduced in Chapter 4 and in this chapter aim to discover recursion schemes in explicitly recursive functions. The approach in Chapter 4 uses *program slicing* as its core technology, where this chapter uses *anti-unification*. The program slicing approach directly inspects the properties of operations in functions, whereas with the anti-unification approach, those properties are implicit in the HOFs in \mathcal{P} . To extend the slicing approach, the programmer must express the desired recursion scheme in terms of how variables are *used* and *updated* between recursive calls. Conversely, the anti-unification approach can be extended by introducing a new HOF implementation. The anti-unification approach does, however, ‘pattern match’ whole functions, often requiring extensive rewriting of the original code. Conversely, the slicing approach inspects individual operations within functions and so requires much less normalisation, if any at all. Finally, the slicing approach is unlikely to misclassify operations due to the order in which arguments are passed to operations, unlike the anti-unification approach, as seen with `squares`, `rowsmult`, and `length` above.

Both approaches have been applied to five parallel benchmarks, and we can discover all `map` operations and pattern instances in \mathcal{P} using our prototype implementations. However, the anti-unification approach can inherently discover a wider range of patterns than the slicing approach. Both approaches have prototype implementations, and synthetic benchmarks suggest that our slicing approach runs in quadratic time with respect to the number of operations, and our anti-unification approach runs in exponential time with respect to the number of function clauses but runs in linear time with respect to the number of functions. Therefore, we can conclude that, for functions with relatively few clauses, the anti-unification approach is quicker than the slicing approach.

5.8 Summary and Discussion

Recursion schemes define specific ways of traversing data structures, and allow properties to be stated, proved, and used [90]. In functional languages, recursion schemes can be implemented as HOFs, and so they can be used to reason about a program's behaviour. For example, a function expressed solely as a `foldr` over an inductive data-type will always terminate [1]. Alternatively, recursion schemes can be used as *loci* for potential parallelism, where `map`, `foldr`, etc. are replaced with equivalent parallel skeletons [19, 31, 107]. Not all possible instances of recursion schemes are guaranteed to be found in code, however. Programmers may define functions as explicitly recursive functions where a recursion scheme may be used. Moreover, when a programmer does define a function as an instance of a recursion scheme, there is no guarantee that the scheme used best expresses the type of traversal; e.g. using a `foldr` when a `map` could also have been used. It follows that a program with more explicit recursion schemes can be better reasoned about in terms of its behaviour, and has more components that can be used in deriving a parallel version of that program.

This chapter presents a methodology to automatically detect and introduce recursion schemes, implemented as HOFs, in Haskell 98 code. The presented approach determines whether a function is expressible as some other function, i.e. an instance of a scheme. While we assume that the scheme implementation is provided, it can, in principle, be generated automatically; e.g. as in the derivable `Functor` class. In Section 5.5, we described a prototype implementation of our approach, where we extended the Haskell refactoring tool, HaRe. Given the assumptions in Section 5.2, the inspected function is anti-unified against a pattern using the algorithm defined in Section 5.3. The results of the anti-unification process are then evaluated to determine whether the inspected function is an instance of the pattern (Section 5.4), and if so, to rewrite the function as a call to the pattern (Section 5.4.1). This approach has been developed and tested for the `fold` family of recursion schemes, where a data structure is traversed recursively. In Section 5.6, we describe modifications to the approach to enable detection of `unfold` recursion schemes, where a data

structure is *constructed corecursively* [100].

In both the traversal and construction cases, the approach that has been presented requires that the inspected function is structurally similar to the pattern that it is being compared to. As a result, the ordering of expressions that are passed to functions, such as fixpoint functions, can result in misclassification. For example, `squares`, `rowsmult`, and `length` will not be rewritten as calls to `foldr`.

```

1 squares [] = []
2 squares (h:t) = h * h : squares t
3
4 rowsmult cols [] = []
5 rowsmult cols (r:rs) =
6   rowmult r cols : rowsmult cols rs
7
8 length [] = 0
9 length (x:xs) = 1 + length xs
10
11 foldr g z [] = z
12 foldr g z (x:xs) = g x (foldr g z xs)

```

In `squares`, the head of the list, `h`, appears twice as an argument to `(*)`. Conversely, in `length`, the head of the list, `x`, does not appear at all. In `rowsmult` the head of the list, `r`, is passed once to the fixpoint function, `rowmult`, but is the *first* passed argument, whereas the head of the list in `foldr` is the *last* passed argument. Refactorings may be used to (semi-)automatically avoid argument ordering misclassification in addition to ensuring other assumptions on the inspected function are met. For example, lambda lifting may be applied in `squares` to ensure `h` is used only once, and again in `rowsmult` to effectively reorder parameters passed to `rowmult`. Similarly, lambda lifting could be used to introduce a lambda that throws its argument away in order to ensure that `x` is used in `length`, without changing the behaviour of the function.

Another, more interesting, result of the structural similarity implicit requirement is that *near patterns* cannot be found. For example, a function that inserts an element into an ordered list,

```
1 insert :: Ord a => a -> [a] -> [a]
2 insert x [] = [x]
3 insert x (y:ys) = if x < y
4                   then x : y : ys
5                   else y : insert x ys
```

can be defined using a `foldr`, but requires additional structure; e.g.:

```
1 insert :: Ord a => a -> [a] -> [a]
2 insert x ys = let Left r = foldl (g x) (Right []) ys
3               in r
4   where
5     g x (Left ys) y = Left (ys ++ [y])
6     g x (Right ys) y = if x < y
7                       then Left (ys ++ [x,y])
8                       else Right (ys ++ [y])
```

Here, the initial value uses the `Either` type to encode additional information, i.e. whether `x` has been inserted to the resulting list. The final result must be extracted from the result of the `fold`, performed here in Lines 2 and 3. We conjecture that the `foldr` in `insert` could be found by clever rewriting of the original definition, but this approach is not ideal since this relies on an external normalisation process; instead the detection of near patterns is deferred to future work.

The modifications that have been made to detect `unfold` could potentially be used to discover composite patterns, e.g. the well-known *map-reduce* pattern, or to detect recursion schemes in mutually recursive functions. Additionally, the approach presented does not attempt discovery of recursion schemes in impure, or monadic, functions since side effects may obscure dependencies, e.g. reading from a file. Further work on the normalisation stage could provide solutions to these problems. Finally, improved normalisation can be used in conjunction with HOF synthesis to produce a range of different, but functionally equivalent, definitions to further improve the chance of discovering a given recursion scheme.

Conclusions and Future Work

In this thesis, we have introduced two novel approaches to automatic pattern discovery for functional languages with the goal of identifying easily parallelisable recursion schemes. This thesis builds upon existing pattern-based and refactoring approaches to parallelism. Our approaches rewrite explicitly recursive functions as instances of recursion schemes that can be used as *loci* of potential parallelism.

6.1 Main Achievements

In Chapter 4, we introduce a novel *program slicing* approach to pattern discovery. Whilst this approach has been defined for a small expression language, it is representative of common functional syntactic elements, and it can be easily compiled to, e.g. Core Haskell, or extended with other syntactic elements. Similarly, whilst the approach is defined for *cons*-lists, it can be easily extended to support arbitrary data types. Since our approach makes no other assumptions, and it inspects individual operations *within* pure explicitly recursive functions, we are able to analyse a wider range of functions than existing techniques. Program calculational approaches require that the entire inspected function is an instance of the desired pattern, and also often require multiple function definitions that traverse data-structures of specific types to derive a divide-and-conquer pattern. Cook’s higher-order unification approach similarly requires that the entire inspected function is an instance of a pattern, and is further

restricted since the approach requires that the inspected function has a very similar structure to the pattern being searched for. By inferring whether variables are *used* or *updated* between successive recursive calls, our approach can work on the expression-, or operation-level as opposed to the function-level. We infer this by defining a novel program slicing algorithm in Section 4.3.2. The greater granularity of our analysis can potentially lead to the discovery of more pattern instances in the same program than these coarse-grained approaches, or discover patterns with potentially less preliminary transformation to facilitate analysis. Kannan similarly avoids this limitation by applying *distillation* prior to pattern discovery [74]. This normalisation stage enables the analysis of functions that could not otherwise be analysed. Unfortunately, since *distillation* is a *deforestation* technique, it can *remove* potential pattern instances in the source code. Consequently, the use of *distillation* prior to pattern discovery can reduce the scope for parallelisation in the transformed code [36, 41]. Conversely, our slicing approach does not perform any deforestation and so will not remove *loci* of potential parallelism. Instead, our approach is able to analyse these functions without normalisation. The approach in [17] also avoids the function-level limitation through the use of a heuristic-based inspection of code. Unfortunately, this approach is limited by the heuristics made available to the approach, and is difficult to extend to other languages. Conversely, our approach is, in principle, language independent for languages where side-effects and state are explicit. Ahn and Han’s program slicing approach [2] is similar to our own approach, but it requires that inspected functions have a particular structure, and that the language is first-order. Conversely, our approach supports both higher-order languages and arbitrary function definitions.

In Chapter 5, we introduce a novel *anti-unification* approach to pattern discovery, including a novel *anti-unification* algorithm in Section 5.3. While this is not the first pattern discovery approach that uses *anti-unification* (i.e. [45]), our approach requires fewer input functions (two instead of four), and can discover a wider range of patterns. Program calculational approaches typically look for a single, general pattern; i.e. homomorphisms or hylomorphisms. Conversely, our approach is *user-extensible*; i.e. it can discover patterns provided by the programmer. This

is possible because we compare the structure of the inspected function and a given pattern. Consequently, our comparison and verification approach in Section 5.4 are generic for all function pairs. Moreover, and unlike higher-order unification, we use first-order anti-unification, which will always produce an anti-unifier for any two functions. These patterns can traverse or construct data-structures of arbitrary types. This includes product types that require simultaneous induction, such as `zipWith`, and corecursive traversals, such as `unfold`. Neither of these patterns can be discovered by the approaches introduced by Cook [41] or Kanan [74], the two existing approaches which can discover the widest range of patterns. Ahn and Han’s program slicing approach [2] is able to discover `zipWith` patterns, but cannot discover `unfold` instances. Nor are any of these approaches user-extensible. Although the approaches by Castro [30] and Chin [36] to discover hylomorphisms in code are inherently able to discover both `fold` and `unfold` instances, these approaches can only discover the more general hylomorphism pattern. Conversely, our approach can discover useful specialisations, such as `map` and `scanl`.

Although both automatic pattern discovery approaches that are introduced in this thesis have been designed to be used to discover *loci* of potential parallelism, these approaches only aim to discover and introduce pattern instances, not to parallelise them. Parallelisation can instead be performed using other techniques, e.g. [107, 30]. Not introducing parallelism immediately has the advantage of exposing information about how the program behaves. The behavioural information encoded by recursion schemes [90] can be used in other domains. For example, the discovery of folds, or specialisations thereof, can facilitate termination checking, the verification of compilers [89], and potentially help the programmer write proofs, particularly in languages such as Agda or Idris. More general program transformations can also benefit, as can compilation, optimisation, and deforestation techniques. Any transformation or static analysis technique that require understanding how data-structures are manipulated throughout the program can benefit from automatic pattern discovery approaches, and the patterns they expose.

6.1.1 Comparison of Approaches

Despite the aim of both above approaches being pattern discovery, their methods and results differ. As part of our evaluation, we applied both slicing and anti-unification approaches to each of the five parallel benchmarks, as well as a subset of the standard `Data.List` Haskell Prelude module. Aside from the slicing approach only seeking to discover `map` instances, both approaches are dependent on a normalisation or translation phase prior to analysis that can affect which patterns are found. Moreover, approaches can either discover the same or different `map` instances depending on the choice of translation and normalisations; which we explore further below.

In our `Data.List` subset, we applied both approaches to 18 functions. Our anti-unification approach was applied to a larger subset, totalling 34 functions, with the intention to evaluate the wider range of patterns discoverable by this approach. Within the subset of 18 functions analysed by both approaches, our slicing approach discovered five *unobstructive* operations in five recursive functions, but our anti-unification approach discovered only three `map` instances. Both slicing and anti-unification approaches discover the `map` instance in the standard Prelude `map` function. The `map` instances discovered in `heads` and `tails` by our anti-unification approach are not found by our slicing approach since a second case expression is used in order to find the head or tail of the inner list; e.g.

```

1 heads :: [[a]] -> [a]
2 heads xss0 =
3   case xss0 of
4     [] -> []
5     (xs:xss) -> case xs of
6                   [] -> []
7                   (y:ys) -> y : (heads xss)

```

Hypothetically, had the standard `head` function been called on `xs`, both approaches would discover the `map` instance. Similarly, our slicing approach discovers *unobstructive* operations in `init`, `intersperse`, and

`last` since all three functions have a null check on the tail of the traversed list in order to compensate for the restrictions of the expression language. For example, given the definition of `last`,

```

1 last xs0 =
2   case xs0 of
3     [] -> []
4     (x:xs) -> if null xs
5                 then x
6                 else last xs

```

The operation `(null xs)` is classified as *unobstructive* since `xs0` is *used* but not *updated*, and so `xs` is considered to be *clean*. In principle, this result allows `(null xs)` to be lifted into a `map` operation, producing a list of boolean values such that only the last element of the list is `True`, given a `map` definition whose fixpoint function can inspect the tail of the list. Conversely, given the normalised `last` definition for our anti-unification approach,

```

1 last (x:xs) = last' x xs
2
3 last' z [] = z
4 last' z (x:xs) = last' (flip const z x) xs

```

the helper function, `last'`, is found to be an instance of `foldr`. Hypothetically, had `last'` been passed to our slicing approach, no *unobstructive* operations would be found due to `z` being both *used* and *updated*. The final function our slicing approach identifies as containing an *unobstructive* operation is `max` due to a ‘mistranslation’, i.e. an incorrect definition.

```

1 max xs0 =
2   case xs0 of
3     [] -> []
4     (x:xs) -> max' xs x
5
6

```

```

7  max' ys0 z =
8    case ys0 of
9      [] -> z
10     (y:ys) -> if gt y ys0
11                  then max' ys y
12                  else max' ys z

```

Here, `gt` in the helper function is *correctly* classified as *unobstructive* despite receiving the wrong argument, `ys0`. Similarly, given the correct arguments, `y` and `z`, `gt` is correctly classified as *obstructive*. Since the normalised definition analysed by our anti-unification approach does not contain this error, it is correctly identified as containing a `foldr` instance.

The differences in results for our parallel benchmarks between the two approaches are for similar reasons. For Sudoku, our slicing approach discovers four potential `map` operations, including the `sudoku` function from Section 4.1. For the equivalent normalised functions, our anti-unification approach discovers the same `map` instances; the remaining `map` instance is identified as a `zipWith` instance, a pattern which can be thought of as a `map` over two lists.

The SumEuler benchmark is comprised of three modules: `ListAux`, `SumEuler`, and `SumEulerPrimes`. Our slicing approach discovers three *unobstructive* operations in three functions defined in `ListAux`. The translated `ListAux` includes a definition of `replicate` that was not normalised for the anti-unification approach since it is normalised in the `Data.List` example subset. Here, the anti-unification approach discovers the same `map` instance. The other two functions that are discovered to contain mappable operations by our slicing algorithm, `takeEach` and `splitIntoN`, are both found to contain `zipWith` instances by our anti-unification approach after normalisation. The `SumEuler` module contains three functions that our slicing approach finds to contain functions with mappable operations. The first, `sumeuler` is identified as a `foldr` instance by our anti-unification algorithm, as in Section 5.1. The `foldr` fixpoint function contains both `(+)` and `euler` operations, whereas the `map` fixpoint function produced by the slicing approach contains only the `euler` operation. The remaining two functions identified

as containing `map` instances by our slicing algorithm were not included in our normalised code that we analysed by our anti-unification algorithm as we chose not to normalise duplicate code (Section 5.7.1). Manual inspection suggests that `sumEuler_JFP`, which traverses a chunked list of inputs, would be identified as a `map` instance. The original definition of `splitIntoChunks` contains two nested list comprehensions, which are equivalent to two `map` operations that we know to be discoverable from our `Data.List` examples. Finally, our slicing approach identifies three functions in `SumEulerPrimes` that have operations which can be lifted into `map` operations. Two of these functions, `phiOpt` and `pair`, operate over tuples. As our expression language does not currently support tuples, `fst` and `snd` functions are used to access the first and second elements respectively. It is these operations that are classified as *unobstructive* by our approach. The equivalent normalisations used for our anti-unification approach result in a single function, which is identified to be a `foldl` instance, capturing the more general behaviour of the functions. Our anti-unification approach discovers the `map` instance in the third function, `primeList`.

Our slicing approach identifies two functions in the N Queens benchmark that contain mappable operations: `safe` and `genloop`. Our anti-unification approach discovers the same `map` in `genloop`, but rewrites `safe` as a `foldl` instead. This `foldl` captures the more general behaviour of `safe`, whereas the `map` produced by our slicing approach captures only one of the logical comparisons. We discussed the case of `safe` in greater detail in Section 5.7.3.

The Matrix Multiplication parallel benchmark comprises two modules: `ListAux`, and `MatMult`. `ListAux` is the same module used for the SumEuler benchmark, which we discussed above. `MatMult` contains two functions that are discovered to contain `map` instances by our slicing approach. Our anti-unification approach discovers the same `map` instances in the equivalent normalisations.

Finally, the N-Body benchmark contains a single function which is identified to contain a `map` instance by our slicing approach. As in the SumEuler benchmark, this function is not normalised in our anti-unification examples due to our normalisation approach (Section 5.7.1).

However, the function identified as a `map` is, in the original code, an explicit `map` operation. We know from the `Data.List` examples that our anti-unification approach can (re)discover unfolded `map` operations.

From the above comparison, it is obvious how normalisation and translation affects the result of both approaches. Translation is primarily affected by the restrictive nature of the expression language our slicing approach uses to analyse functions. We see from the `heads` and `tails` examples from `Data.List` that this can result in no `map` operation being discovered since our slicing approach does not consider the syntactic structure of the whole function. Our anti-unification approach, which specifically *does* inspect the syntactic structure of functions, is reliant on more extensive normalisations. According to these normalisations, the anti-unification approach can discover the same `map` instances as our slicing approach, but can also potentially discover a wider range of pattern instances for the same program.

The approaches presented in the theses of Cook and Kannan are similar to the anti-unification approach presented in this thesis. In order to compare our anti-unification approach with their techniques, we applied our approach to examples taken from their theses. We looked at all of Cook's examples, for which we took the decision to normalise a single function multiple times in order to demonstrate that we can find potentially multiple patterns for a single function. In all of Cook's examples our approach is able to discover the same `map`, `foldr`, `foldl`, and `scan` patterns. Our approach also discovers additional `foldr` instances. This is in part due to all `map` instances being normalised as `foldr` instances. Further additional `foldr` instances are discovered in `mmult` and `mergesort` examples as we apply our approach to functions that Cook did not. These include `dist` and `dotproduct` in `mmult`, and `merge` in `mergesort`. All three functions are instances of `foldr` over *two* lists, and act as examples of *simultaneous induction* that Cook's approach cannot inspect [41].

We applied our approach to a representative subset of Kannan's examples. The subset does not include: his problem cases, his Fibonacci Series sum, or his Sum Squares of List examples. The Sum Squares of List example is equivalent to our `sumeuler` example, where `euler` is

replaced with a square operation. Similarly, his Fibonacci Series Sum is another simple `foldr` case akin to his Dot Product or Power Tree examples, both of which we have applied our approach to. Kannan's problem cases include: Maximum Segment Sum, Reverse List, Flatten Binary Tree, and Insertion Sort. The original code for all four examples use some combination of `map`, `inits`, `max`, `tails`, and the append operation `(++)`, all of which we have demonstrated to be discoverable in our `Data.List` examples. We also demonstrated that we are able to discover one `unfold` instance in Insertion Sort in Section 5.7.5. For the remaining examples, we have normalised both the original programs, and where possible, the distilled programs. This demonstrates that it is possible to combine our approach with a deforestation technique if so desired. As Kannan directly introduces skeletons into the code, we assume, derived from his own definitions, that `farmB` is a `map`, `offlineParMapRedr` is a combination of a `map` and a `foldr`, and `accumulate` is a combination of a `scan` and a `foldr`.

In Kannan's Matrix Multiplication example, we discover two `foldr` instances that correspond to Kannan's two discovered `map` instances, and a `foldl` that corresponds to the map-reduce's `foldr` instance. We derive a leftwards-fold due to the structure of the function inspected, but the reduce skeleton implies that *either* a leftwards- or rightwards-fold can be used. The discovery of `foldr` instances instead of `map` instances is valid, but produces a less specialised result. A similar lack of specialisation is found in Kannan's Totient example, where our approach discovers the `foldr` instance but does not discover a combination of a `map` and a `foldr` due to less vigorous normalisation. Our anti-unification approach discovers the same `foldr` and `scan` instances discovered by Kannan in all other examples; a total of one `scan` instance and three `foldr` instances. In Kannan's Maximum Prefix Sum example, our approach finds an additional `foldr` and an `unfold` instance from the distilled definitions.

The above comparisons demonstrate that our approach can potentially discover more patterns than Cook's approach, including instances of simultaneous induction and `unfold` patterns. Kannan's approach benefits from a more aggressive normalisation approach in some examples, but

our anti-unification approach is capable of discovering either the same or equivalent pattern instances in the same functions, as well as additional `unfold` operations.

6.1.2 Limitations

Both of the approaches presented in this thesis have a number of limitations, which we describe in this section. One limitation that is shared by both approaches is that they require purity; we are unable to (safely) analyse and rewrite effectful code. This can present problems when effectful code is still parallelisable; e.g. a `map` operation that accesses independent elements of a file or shared data structure. While both approaches can, in principle, be adapted to work with languages other than Haskell, our approaches have not yet been tested with other languages, including ones with implicit effects such as C/C++. We speculate that adapting to other pure functional languages should not be difficult, but adapting our approach to impure functional languages, such as OCaml or Erlang, imperative languages, or effect-oriented languages would require significantly more work since side-effects can obfuscate both changes to state and dependencies. It may be possible, e.g. as in the polyhedral model, to increase the granularity of information to inspect how data structures, and potentially effectful state, are accessed. Alternatively, dynamic analyses could be used to gather this information. The ability to handle effectful programs will allow application of our approaches to a wider range of programs, and thereby facilitate and further automate the parallelisation of these programs.

Limitations of the Slicing Approach

The primary limitation of the slicing approach that we have shown here is that it can only discover and introduce *map* operations over *cons*-lists. In principle, with an additional refactoring, it is possible to introduce `zipWith` pattern instances to functions that exhibit simultaneous induction. Similarly, since the refactoring that we defined in Section 4.5 introduces `map` operations only in structurally recursive functions, the helper function that is also introduced may be considered to be a `fold`.

However, and unlike the introduced `map` operation, neither the fixpoint function nor the initial value of the implicit `fold` instance is discovered, and it would require additional work to infer them, perhaps in a similar approach to argument inference in the anti-unification approach.

Expanding the approach to other patterns and types has three requirements: *i)* defining the concept of *update* for each type, *ii)* defining the pattern in terms of *usage* and *update* of function arguments between successive recursive calls, and *iii)* defining pattern-specific refactorings. Currently, our approach permits only three types: booleans, integers, and *cons*-lists. Consequently, all programs and all types, e.g. binary trees, must be converted to these three types in order for our analysis to succeed. This limitation may be avoided by extending the expression language to allow arbitrary types and constructors. It would then be necessary to redefine *usage* and *update*. A generic definition could be provided, and it could then be augmented by specialisations for certain types. This would allow functions that traverse arbitrary data-structures without first converting to lists, and it would therefore simplify the application of our approach to a wider range of programs. Since our approach is defined to discover `map` operations, we have necessarily defined (un)obstructiveness in such a way that the combination of variables that are *used* and *updated* reflects the property of the `map` pattern. It follows that to discover additional patterns, each pattern must be expressed as a combination of *usage* and *update* categorisations of variables. This may not even be possible for all patterns. Finally, we observe that the result of the slice and analysis merely detects the computations that can be performed as part of a fixpoint function that is passed to a pattern. It relies upon a refactoring to use this information and rewrite the inspected function according to whichever pattern is being detected. It follows that for each pattern that our slicing approach is able to detect, a refactoring must also be defined to introduce that pattern. This may be non-trivial, in particular, e.g., for programmers working with a custom skeleton that wish to detect and introduce that pattern automatically.

The slicing approach is also limited in its present form since the definitions of *usage* and *update* are relatively coarse. The Smith-Waterman Example in Section 4.6.3 suggests that a finer understanding of how each

element in the list is *used* and *updated* may provide greater insight as to which operations are performed independently of other elements. For example, inferring how elements in a data structure are accessed between recursive calls could expose groups of accesses that are independent of all other accesses over successive recursive calls. Finer-grained definitions of *usage* and *update* would enable the discovery of a wider range of patterns, and thus more opportunities for parallelism. It could also facilitate expanding the approach to other schemes and effectful code.

Limitations of the Anti-Unification Approach

The first and most major limitation of the anti-unification approach is that it relies heavily upon the normalisation process to successfully discover instances of recursion schemes. This assumes that any normalisation process is sound, and that it is able to discover definitions that are structurally similar to a given recursion scheme. This is compounded further by the inflexibility of our current approach to argument inference. For example, given the function `squares`,

```
1 squares [] = []
2 squares (x:xs) = x*x : squares xs
```

our approach will not discover the obvious `map` instance, since `x` is used twice as an argument to `(*)`. A more flexible approach to argument inference would reduce the dependence on normalisation, albeit not eliminate it. Extending equivalence checking with equational reasoning could reduce this dependence further. Reducing the need for normalisation would also simplify the discovery of multiple schemes in a single inspected function, e.g. discovering both `map` and `foldr`, but also *combinations* of schemes, e.g. the *map-reduce* pattern.

Another source of inflexibility is that whilst we are able to discover arbitrary schemes that are expressible as an instance of `foldr`, our approach may not easily discover schemes with different structures. As demonstrated in Section 5.6, our approach could be used as the basis of a more general analysis and pattern discovery technique. Similarly, addressing this limitation could facilitate the discovery of schemes with

arbitrary structures. The ability to detect patterns with arbitrary structures would allow the detection of a wider range of schemes, and so ensure that as many *loci*, or components, are available for parallelisation.

Finally, one minor limitation is that our current prototype implementation is built as an extension of old software. This limits ease of use and distribution. It should, however, be straightforward to reimplement our prototype using the latest version of HaRe, which itself uses GHC's AST.

6.2 Future Work

The work presented in this thesis can be carried forward in a number of directions in order to overcome the limitations described above. We could improve the slicing approach by extending our expression language and definitions to support arbitrary types. We could then expand the range of schemes that are discoverable using this approach. Both expansions would allow the slicing approach to be applied to a wider range of programs, and expose more *loci* of potential parallelism. We also intend to explore approaches that infer how elements are accessed in data-structures that are traversed by inspected functions, perhaps by an extension of our definitions of *usage* and *update*. To address the difficulty of encoding new schemes in terms of *usage* and *update*, thus enabling their discovery as part of our slicing approach, we intend to investigate how schemes might be encoded automatically. This would allow programmers to extend our approach with custom skeletons and their equivalent recursion schemes.

To improve our anti-unification approach, we intend to address its dependence on normalisation. We intend to incorporate equational reasoning techniques in order to improve argument derivation, and testing the equivalence of anti-unifier and a given recursion scheme. This will facilitate the application of our approach to larger and real-world programs, and likely result in the discovery of more recursion scheme instances. Relatedly, we intend to extend our approach to work with recursion schemes with arbitrary structures. We could first improve detection of `unfold` patterns. This will allow the discovery of a wider range of patterns, e.g.

hylomorphisms. Finally, we intend to investigate how implementations of recursion schemes could be generated automatically. Since recursion schemes can be implemented in multiple ways, this could increase the chances that a scheme is found. Automatic pattern generation could also mean that implementations would not need to be provided.

We intend to test both approaches against larger and real-world programs. This will provide a better understanding of the limitations of our approaches, and potentially indicate how they might be improved. Similarly, and since both approaches are, in principle, language agnostic, we intend to adapt and apply our techniques to languages other than Haskell. Finally, but importantly, we could first apply our approaches to impure functional languages, e.g. Erlang. This would allow us to extend both approaches to inspect impure code, and so increase the range of functions to which they can be applied. This would also facilitate the application of our approaches to imperative languages such as C++ and Java.

6.3 Concluding Remarks

This thesis has introduced two novel approaches to automatic pattern discovery for pure functional languages. Our *program slicing* approach is, in principle, able to analyse a wider range of pure explicitly recursive functions than current approaches. Similarly, our *anti-unification* approach is able to discover a wider range of patterns, including custom patterns provided by the programmer, in pure explicitly recursive functions than current approaches. The discovery of more pattern instances, and a wider variety of patterns, facilitates the introduction of parallelism by exposing more opportunities for parallelism and by further automating the process. Consequently, it is now possible to automatically parallelise more functional programs with a wider range of parallel structures. This allows programmers to take advantage of now-ubiquitous parallel hardware in a simple and safe way, and improve the performance of their programs with minimal effort.

Proof for Soundness of Slicing Algorithm

Lemma A.1. *For all programs, p , and their respective program environments, Γ_p ; for all fixpoint expressions, e , in p where,*

$$e = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow e')$$

and for all variables x , such that $x = x_\mu$ where $\mu \in [1, n]$, we can derive the slice of e with criterion x , $\Sigma^{e|x}$, by

$$\Gamma_p, f, x, \mu \vdash e : \Sigma^{e|x}$$

using the inference rules in Figure 4.4. Similarly, for all subexpressions, η, δ , of e , (i.e. $\forall \eta, \eta \ll e$ and $\forall \delta, \delta \ll e$) we obtain the slice of η , $\Sigma^{\eta|x}$, by

$$\Gamma_p, f, x, \mu \vdash \eta : \Sigma^{\eta|x}$$

and the slice of δ by

$$\Gamma_p, f, x, \mu \vdash \delta : \Sigma^{\delta|x}$$

Given both $\Sigma^{\eta|x}$ and $\Sigma^{\delta|x}$, when $\delta \ll \eta$, it follows that

$$\forall y, y \in \Sigma^{\delta|x} \text{ implies that } y \in \Sigma^{\eta|x}$$

Proof Sketch. The proof is by structural induction on η .

Case 1. $\eta = y$, given that $y \triangleleft_p^* x$.

Since $y \triangleleft_p^* x$, the inference rule VAR_1 applies and so, $\Sigma^{y|x} = \{y\}$.

Since y is a variable, from the definition of subexpression (Definition 4.2.2) we know that $\delta = \eta = y$.

Consequently, $\Sigma^{\delta|x} = \Sigma^{y|x}$.

It follows trivially that y is in the slice of both δ and η since $\delta = \eta = y$ and $\Sigma^{\delta|x} = \Sigma^{\eta|x} = \Sigma^{y|x}$.

Case 2. $\eta = \text{cons}_\tau z \, zs$, given that $\eta \equiv x$; i.e. that $z \triangleleft_p x$, $zs \triangleleft_p x$, and $z \neq zs$.

Since $\eta \equiv x$, the inference rule LST_2 applies and so, $\Sigma^{\eta|x} = \{x, z, zs\}$.

From the definition of subexpression (Definition 4.2.2), there are three subcases.

Case 2.1. When $\delta = \eta$.

This case is analogous to Case 1.

Case 2.2. When $\delta = z$.

Since $\delta = z$, and since we know that $z \triangleleft_p^* x$ by Definition 4.2.3, the inference rule VAR_1 applies and so, $\Sigma^{z|x} = \{z\}$.

It follows trivially that both $z \in \Sigma^{z|x}$ and $z \in \Sigma^{\delta|x}$.

Case 2.3. When $\delta = zs$.

This case is analogous to Case 2.2.

Case 3. $\eta = \text{cons}_\tau e_1 \, e_2$, given that $\eta \not\equiv x$.

Since $\eta \not\equiv x$, the inference rule LST_3 applies and so, $\Sigma^{\eta|x} = \Sigma_1 \cup \Sigma_2$.

From the definition of subexpression (Definition 4.2.2), there are three subcases.

Case 3.1. When $\delta = \eta$.

This case is analogous to Case 1.

Case 3.2. When $\delta = e_1$.

The induction hypothesis (IH) is $y \in \Sigma^{e_1|x}$.

Given $\Gamma_p, f, x, \mu \vdash e_1 : \Sigma^{e_1|x}$, $\Gamma_p, f, x, \mu \vdash e_2 : \Sigma^{e_2|x}$, and IH, it follows trivially from $\Sigma^{\eta|x} = \Sigma_1 \cup \Sigma_2$ that $y \in \Sigma^{\eta|x}$.

Case 3.3. When $\delta = e_2$.

The case is analogous to Case 3.2.

Case 4. $\eta = \text{case } ys \text{ of } \text{nil}_\tau \rightarrow e_1, \text{cons}_\tau z zs \rightarrow e_2$.

This case is analogous to Case 3.

Case 5. $\eta = f(e_1, \dots, e_n)$.

The inference rule REC-APP applies and so, $\Sigma^{\eta|x} = \bigcup_{j \in [1, n]} \Sigma_j$.

From the definition of subexpression (Definition 4.2.2), there are n subcases; i.e. $\delta = e_j$. Since REC-APP defines a special case for e_μ , these subcases can be divided into two groups: where $j \neq \mu$ and $j = \mu$.

Case 5.1. Where $j \neq \mu$.

This case is similar to Case 3.2.

The induction hypothesis is $y \in \Sigma^{e_j|x}$.

Given the slice for each e_j , $\Sigma^{e_j|x} = \Sigma_j$, according to REC-APP; since $\Sigma^{\eta|x} = \bigcup_{j \in [1, n]} \Sigma_j$; and from the induction hypothesis we know that $\Sigma_\mu = \emptyset$; it follows trivially that $y \in \Sigma^{\eta|x}$.

Case 5.2. Where $j = \mu$.

The REC-APP inference rule defines four special cases for e_j , consequently there are four subcases.

Case 5.2.1. Where $e_\mu \triangleleft_p^* x$.

From the REC-APP inference rule, $\Sigma_\mu = \emptyset$.

It follows trivially that since $\Sigma^{\delta|x} = \Sigma_\mu = \emptyset$, $y \in \emptyset$ implies that $y \in \Sigma^{\eta|x}$.

Case 5.2.2. Where $e_\mu \equiv x$.

This case is analogous to Case 5.2.1.

Case 5.2.3. Where $e_\mu = \text{cons}_\tau e_k e_l$, given that $e_l \equiv x$ and that there are no subexpressions, $e'_{k'}$, of e_k that are either reachable from x or a recursive call (i.e. $\neg(\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn})))$).

This case is analogous to Case 5.2.1.

Case 5.2.4. Otherwise, where $\neg(e_\mu \triangleleft_p^* x) \wedge (e_\mu \not\equiv x) \wedge \neg(e_\mu = \text{cons}_\tau e_k e_l \wedge e_l \equiv x \wedge \neg(\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn}))))$.

This case is similar to Case 3.2.

The induction hypothesis is $y \in \Sigma_\mu$. Given the slice for e_μ , $\Sigma^{e_\mu|x}$, from REC-APP we know that $\Sigma^{\delta|x} = \Sigma_\mu = \{x\} \cup \Sigma^{e_\mu|x}$.

Since we know that $\Sigma^{\eta|x} = \bigcup_{j \in [1, n]} \Sigma_j$; and from the induction hypothesis we know that $y \in \Sigma_\mu$; it follows trivially that $y \in \Sigma^{\eta|x}$.

Case 6. $\eta = e_0(e_1, \dots, e_n)$, where $e_0 \neq f$.

This case is similar to Case 3.

Since $e_0 \neq f$, the inference rule APP applies and so, $\Sigma^{\eta|x} = \bigcup_{j=0}^n \Sigma^{e_j|x}$.

From the definition of subexpression (Definition 4.2.2), there are $n + 2$ subcases; i.e. $\delta = \eta$ and $\forall j \in [0, n], \delta = e_j$. As before, we group the latter $n + 1$ subcases into a single subcase.

Case 6.1. Where $\delta = \eta$.

This case is analogous to Case 1.

Case 6.2. Where $\forall j \in [0, n], \delta = e_j$.

This case is similar to Case 5.1.

The induction hypothesis is $y \in \Sigma^{e_j|x}$.

It follows trivially that since $\Sigma^{\eta|x} = \bigcup_{j=0}^n \Sigma^{e_j|x}$, and given the induction hypothesis, $y \in \Sigma^{\eta|x}$.

Case 7. $\eta = \lambda(a_1, \dots, a_n) \rightarrow e_1$.

From the FUN inference rule we know that $\Sigma^{\eta|x} = \Sigma^{e_1|x}$.

From the definition of subexpression (Definition 4.2.2), there are two subcases.

Case 7.1. When $\delta = \eta$.

This case is analogous to Case 1.

Case 7.2. When $\delta = e_1$.

The induction hypothesis is $y \in \Sigma^{e_1|x}$.

Given the slice of e_1 , $\Sigma^{e_1|x}$, given the induction hypothesis, and since we know that $\Sigma^{\eta|x} = \Sigma^{e_1|x}$, it follows trivially that $\Sigma^{\eta|x} = \Sigma^{\delta|x} = \Sigma^{e_1|x}$.

Consequently, $y \in \Sigma^{\eta|x}$.

Case 8. $\eta = \text{fix } e_1$.

This case is analogous to Case 7. □

Corollary A.1. *It immediately follows from Lemma A.1 that $\forall y, y \in \Sigma^{\eta|x}$ implies that $y \in \Sigma^{e|x}$.*

Lemma A.2. For all programs, p , and their respective program environments, Γ_p ; for all fixpoint expressions, e , in p where,

$$e = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow d)$$

Definition 4.3.2 produces the variable usage escaped e , $e^\varepsilon = (e \setminus_\varepsilon x)$. For clarity, we will use ε to denote the subexpression corresponding to the same subexpression in e^ε . For example, in

$$e^\varepsilon = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow d^\varepsilon)$$

d^ε corresponds to d in e .

For all subexpressions, η , to e and all subexpressions, δ , to e^ε (i.e. $\forall \eta, \eta \ll e$ and $\forall \delta, \delta \ll e^\varepsilon$), given that η and δ are corresponding subexpressions in e and e^ε respectively (i.e. $\delta = \eta^\varepsilon$), and given the slices of η and δ ,

$$\Gamma_p, f, x, \mu \vdash \eta : \Sigma^{\eta|x} \tag{A.1}$$

$$\Gamma_p, f, x, \mu \vdash \delta : \Sigma^{\delta|x} \tag{A.2}$$

it follows that

$$\Sigma^{\eta|x} = \Sigma^{\delta|x}$$

Proof Sketch. The proof is by case analysis on η .

Recall that the definition of variable usage escapement (Definition 4.3.2) substitutes occurrences of x and expressions that are syntactically equivalent to x in e for the variable ε , representing an empty expression and considered to be syntactically equivalent to x (i.e. $\varepsilon \equiv x$). Definition 4.3.2 targets such substitutable expressions only in recursive calls, specifically the μ^{th} argument, e_μ . Consequently, there are four cases.

Case 1. $\eta = f(e_1, \dots, e_n)$ and $e_\mu = x$, given that $\forall j \in ([1, \mu) \cup (\mu, n])$, $\Sigma^{e_j|x} = \Sigma^{e_j^\varepsilon|x}$.

Since η is a recursive call, the REC-APP inference rule applies and so we know that, $\Sigma^{e_\mu|x} = \Sigma_\mu = \emptyset$.

From the definition of variable usage escapement (Definition 4.3.2), we know that $e_\mu^\varepsilon = \varepsilon$.

Since δ is a recursive call and $e_\mu^\varepsilon = \varepsilon$, the REC-APP inference rule applies and so we know that, $\Sigma^{e_\mu^\varepsilon|x} = \Sigma_\mu = \emptyset$.

Since both η and δ are recursive calls, it follows from the REC-APP rule that $\Sigma^{\eta|x} = \bigcup_{j \in [1, n]} \Sigma_j$ and $\Sigma^{\delta|x} = \bigcup_{j \in [1, n]} \Sigma_j$ where $\Sigma_j = \Sigma^{e_j^\varepsilon|x}$ and where $\Sigma_\mu = \emptyset$.

Since $\forall j \in ([1, \mu) \cup (\mu, n])$, $\Sigma^{e_j|x} = \Sigma^{e_j^\varepsilon|x}$ holds, and that $\emptyset = \emptyset$, it follows trivially that $\Sigma^{\eta|x} = \Sigma^{\delta|x}$.

Case 2. $\eta = f(e_1, \dots, e_n)$ and $e_\mu \equiv x$, given that $\forall j \in ([1, \mu) \cup (\mu, n])$, $\Sigma^{e_j|x} = \Sigma^{e_j^\varepsilon|x}$.

This case is analogous to Case 1.

Case 3. $\eta = f(e_1, \dots, e_n)$ and $e_\mu = \text{cons}_\tau e_k e_l$, given that $e_l \equiv x \wedge \neg(\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn})))$ and $\forall j \in ([1, \mu) \cup (\mu, n])$, $\Sigma^{e_j|x} = \Sigma^{e_j^\varepsilon|x}$.

This case is analogous to Case 1.

Case 4. Otherwise.

From Definition 4.3.2 we know that $\eta = \delta$, and the result follows trivially. \square

Corollary A.2. *It immediately follows from Lemma A.2 that $\Sigma^{e|x} = \Sigma^{e^\varepsilon|x}$.*

Theorem A.1 (Soundness of Slicing Algorithm). *For all programs, p , and their respective program environments, Γ_p ; for all fixpoint expressions, e , in p where,*

$$e = \text{fix}(\lambda(f, x_1, \dots, x_n) \rightarrow e')$$

and for all variables x , such that $x = x_\mu$ where $\mu \in [1, n]$, we can derive the slice of e with criterion x , $\Sigma^{e|x}$, by

$$\Gamma_p, f, x, \mu \vdash e : \Sigma^{e|x}$$

It follows that

$$\forall y, y \in \Sigma^{e|x} \text{ implies that } y \sim_x e$$

where $y \sim_x e$ denotes that y is either used or updated in e according to Definitions 4.3.3 and 4.3.1, respectively.

Proof Sketch. The proof is by case analysis on y .

We must show that if $y \sim_x e$, i.e. if y is either *used* or *updated* in e , then y is in the slice of e , $\Sigma^{e|x}$. From Definitions 4.3.1 and 4.3.3, there are three possible cases. We must also show that if y is neither *used* nor *updated* in e then y is not in $\Sigma^{e|x}$.

Case 1. $y \triangleleft_p^+ x$; i.e. $(y \triangleleft_p^* x) \wedge (y \neq x)$.

From the definition of usage, in order for $y \sim_x x$ to hold, we must show that both $y \triangleleft_p^* x$ and $y \ll (e \setminus_\varepsilon x)$ hold.

Since $y \triangleleft_p^+ e$, it follows trivially that $y \triangleleft_p^* x$.

Let $e^\varepsilon = (e \setminus_\varepsilon x)$, we must show that $y \in \Sigma^{e|x}$ implies that $y \ll e^\varepsilon$.

When y is an expression, we know that the slice, $\Gamma_p, f, x, \mu \vdash y : \Sigma^{y|x}$, produced by the VAR_1 inference rule is $\Sigma^{y|x} = \{y\}$. Consequently, $y \in \Sigma^{y|x}$.

If $y \in \Sigma^{y|x}$ and $y \ll e^\varepsilon$, we know that $y \in \Sigma^{e^\varepsilon|x}$ by Lemma A.1.

If $y \in \Sigma^{e|x}$, we know that $y \in \Sigma^{e^\varepsilon|x}$ by Lemma A.2.

Consequently, $y \in \Sigma^{e|x}$ implies that $y \ll e^\varepsilon$.

Case 2. $y = x$.

From the definition of usage, there are two subcases.

Case 2.1. When $x \ll e^\varepsilon$, given that $e^\varepsilon = (e \setminus_\varepsilon x)$.

This case is analogous to Case 1.

Case 2.2. When $\exists \eta, (\eta \ll e^\varepsilon) \wedge \eta \equiv x$, given that $e^\varepsilon = (e \setminus_\varepsilon x)$.

This case is similar to Case 1, except that the LST_2 inference rule applies instead of VAR_1 .

Case 3. $y = \bar{x}$.

From the definition of update (Definition 4.3.1), $\bar{x} \sim_x x$ holds when there exists an expression η such that $\eta \ll e$ and $\eta = f(e_1, \dots, e_n)$, where $\neg(e_\mu \triangleleft_p^* x) \wedge (e_\mu \not\equiv x) \wedge \neg(e_l \equiv x \wedge \neg(\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn}))))$. Since $\bar{x} \in \Sigma^{\eta|x}$, and from the REC-APP inference rule, we know that $\bar{x} \in \Sigma^{\eta|x}$ when $\neg(e_\mu \triangleleft_p^* x) \wedge (e_\mu \not\equiv x) \wedge \neg(e_l \equiv x \wedge \neg(\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn}))))$ holds.

By Lemma A.1, we know that when $\eta \ll e$ and $\bar{x} \in \Sigma^{\eta|x}$, $\bar{x} \in \Sigma^{e|x}$.

Consequently, $\bar{x} \in \Sigma^{e|x}$ implies that $\bar{x} \sim_x e$.

Case 4. $\neg(y \sim_x e)$.

By inspection, the rules that introduce elements into the slice are VAR_1 , LST_2 , and REC-APP .

It follows from the conditions of VAR_1 , that $y \triangleleft_p^* x$ and therefore that y is *used* in e by definition.

It follows from the conditions of LST_2 , that $\exists \eta, \eta \ll e$ and $\eta = \text{cons}_\tau z z s$ and that $\eta \equiv x$, therefore x, z , and zs are all *used* in e by definition.

If follows from the conditions of REC-APP, that $\exists \eta, \eta \ll e$ and $\eta = f(e_1, \dots, e_n)$ and $\neg(e_\mu \triangleleft_p^* x) \wedge (e_\mu \not\equiv x) \wedge \neg((e_\mu = \text{cons}_\tau e_k e_l \wedge e_l \equiv x \wedge \neg(\exists e'_k, e'_k \ll e_k \wedge (e'_k \triangleleft_p^* x \vee e'_k = f(e'_{k1}, \dots, e'_{kn})))$, therefore \bar{x} is *updated* in e by definition.

□

Proof for Soundness of Anti-Unification Algorithm

Theorem B.1 (Soundness of Anti-Unification Algorithm). *Given the terms t_1 and t_2 , we can find t , σ_1 , and σ_2 , such that $t_1 \cong t \sigma_1$ and $t_2 \cong t \sigma_2$.*

Proof Sketch. Given the terms t , t_1 , t_2 , and substitutions, σ_1 and σ_2 . The proof is by case analysis on t_1 and t_2 .

Case $t_1 \cong t_2$:

By application of EQ with $t = t_2$ and $\sigma_1 = \sigma_2 = \varepsilon$, we must show that $t_1 \cong t_2 \varepsilon$ and $t_2 \cong t_2 \varepsilon$.

By Definition 5.3.2 and reflexivity, both $t_1 \cong t_2$ and $t_2 \cong t_2$ hold.

Case $t_2 = (\mathbf{Var} \ v) \wedge (v \neq \mathbf{p})$:

By application of VAR when $t = (\mathbf{Var} \ v)$, $\sigma_1 = \Sigma v t_1$, and $\sigma_2 = \varepsilon$, we must show that $t_1 \cong (\mathbf{Var} \ v) (\Sigma v t_1)$ and $(\mathbf{Var} \ v) \cong (\mathbf{Var} \ v) \varepsilon$ hold.

By Definition 5.3.1 and reflexivity, $t_1 \cong t_1$ holds.

By Definition 5.3.2 and reflexivity, $(\mathbf{Var} \ v) \cong (\mathbf{Var} \ v)$ holds.

Case $t_1 = (\mathbf{Var} \ v_{a_i}^f) \wedge t_2 = (\mathbf{App} \ (\mathbf{Var} \ u) \ (\mathbf{Var} \ v_{a_i}^p)) \wedge (u \neq p) \wedge (v_{a_i}^f \equiv v_{a_i}^p)$:

By application of ID where

$$\begin{aligned} t &= (\mathbf{App} \ (\mathbf{Var} \ u) \ (\mathbf{Var} \ v_{a_i}^p)) \\ \sigma_1 &= \langle \Sigma vid, \varepsilon \rangle \\ \sigma_2 &= \varepsilon \end{aligned}$$

we must show that:

$$(\mathbf{Var} \ v_{a_i}^f) \cong (\mathbf{App} \ (\mathbf{Var} \ u) \ (\mathbf{Var} \ v_{a_i}^p)) \langle \Sigma uid, \varepsilon \rangle \quad (1)$$

$$(\mathbf{App} \ (\mathbf{Var} \ u) \ (\mathbf{Var} \ v_{a_i}^p)) \cong (\mathbf{App} \ (\mathbf{Var} \ u) \ (\mathbf{Var} \ v_{a_i}^p)) \varepsilon \quad (2)$$

For (1):

By Definition 5.3.3, $(\mathbf{Var} \ v_{a_i}^f) \cong (\mathbf{App} \ (\mathbf{Var} \ u) \ \Sigma uid \ (\mathbf{Var} \ v_{a_i}^p) \varepsilon)$.

By Definition 5.3.1 and Definition 5.3.2,

$$(\mathbf{Var} \ v_{a_i}^f) \cong (\mathbf{App} \ (\mathbf{Var} \ id) \ (\mathbf{Var} \ v_{a_i}^p)).$$

By Definition 5.3.5 and reflexivity, $(\mathbf{Var} \ v_{a_i}^f) \cong (\mathbf{Var} \ v_{a_i}^p)$ holds.

For (2):

By Definition 5.3.2 and reflexivity,

$$(\mathbf{App} \ (\mathbf{Var} \ u) \ (\mathbf{Var} \ v_{a_i}^p)) \cong (\mathbf{App} \ (\mathbf{Var} \ u) \ (\mathbf{Var} \ v_{a_i}^p)) \text{ holds.}$$

Case $t_2 \in \mathcal{R}_p$:

By application of RP where $t = (\mathbf{Var} \ \alpha)$, $\sigma_1 = \Sigma \alpha t_1$, and $\sigma_2 = \Sigma \alpha t_2$, we must show that both $t_1 \cong (\mathbf{Var} \ \alpha) \Sigma \alpha t_1$ and $t_2 \cong (\mathbf{Var} \ \alpha) \Sigma \alpha t_2$ hold.

By Definition 5.3.1 and reflexivity, both $t_1 \cong t_1$ and $t_2 \cong t_2$ hold.

Case $t_1 = (C \ t_{11} \ \dots \ t_{1n}) \wedge t_2 = (C \ t_{21} \ \dots \ t_{2n}) \wedge$

$\forall i \in [1, n], (t_{1i} \cong t_i \sigma_{1i}) \wedge (t_{2i} \cong t_i \sigma_{2i})$:

By application of CONST where

$$\begin{aligned} t &= (C \ t_1 \ \dots \ t_n) \\ \sigma_1 &= \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \\ \sigma_2 &= \langle \sigma_{21}, \dots, \sigma_{2n} \rangle \end{aligned}$$

we must show that:

$$\begin{aligned} (C \ t_{11} \ \dots \ t_{1n}) &\cong (C \ t_1 \ \dots \ t_n) \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \\ (C \ t_{21} \ \dots \ t_{2n}) &\cong (C \ t_1 \ \dots \ t_n) \langle \sigma_{21}, \dots, \sigma_{2n} \rangle \end{aligned}$$

By Definition 5.3.3, $(C \ t_{11} \ \dots \ t_{1n}) \cong (C \ (t_1 \sigma_{11}) \ \dots \ (t_n \sigma_{1n}))$
and $(C \ t_{21} \ \dots \ t_{2n}) \cong (C \ (t_1 \sigma_{21}) \ \dots \ (t_n \sigma_{2n}))$.

By assumption, substitutivity and reflexivity,

both $(C \ t_{11} \ \dots \ t_{1n}) \cong (C \ t_{11} \ \dots \ t_{1n})$ and $(C \ t_{21} \ \dots \ t_{2n}) \cong (C \ t_{21} \ \dots \ t_{2n})$
hold.

Case $t_1 = [t_{11}, \dots, t_{1n}] \wedge t_2 = [t_{21}, \dots, t_{2n}] \wedge$
 $\forall i \in [1, n], t_{1i} \cong t_i \sigma_{1i} \wedge t_{2i} \cong t_i \sigma_{2i}$:

By application of LIST where

$$\begin{aligned} t &= [t_1, \dots, t_n] \\ \sigma_1 &= \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \\ \sigma_2 &= \langle \sigma_{21}, \dots, \sigma_{2n} \rangle \end{aligned}$$

we must show that:

$$\begin{aligned} ([t_{11} \ \dots \ t_{1n}]) &\cong ([t_1 \ \dots \ t_n]) \langle \sigma_{11}, \dots, \sigma_{1n} \rangle \\ ([t_{21} \ \dots \ t_{2n}]) &\cong ([t_1 \ \dots \ t_n]) \langle \sigma_{21}, \dots, \sigma_{2n} \rangle \end{aligned}$$

By Definition 5.3.3, $([t_{11} \ \dots \ t_{1n}]) \cong ([t_1 \ \dots \ t_n]) \langle \sigma_{11}, \dots, \sigma_{1n} \rangle$
and $([t_{21} \ \dots \ t_{2n}]) \cong ([t_1 \ \dots \ t_n]) \langle \sigma_{21}, \dots, \sigma_{2n} \rangle$.

By assumption, substitutivity and reflexivity,

both $([t_{11} \ \dots \ t_{1n}]) \cong ([t_{11} \ \dots \ t_{1n}])$ and $([t_{21} \ \dots \ t_{2n}]) \cong ([t_{21} \ \dots \ t_{2n}])$
hold.

Otherwise:

By application of OTHERWISE where $t = (\text{Var } \alpha)$, $\sigma_1 = \Sigma \alpha t_1$, and $\sigma_2 = \Sigma \alpha t_2$, we must show that both $t_1 \cong (\text{Var } \alpha) \Sigma \alpha t_1$ and $t_2 \cong (\text{Var } \alpha) \Sigma \alpha t_2$ hold.

By Definition 5.3.1 and reflexivity, both $t_1 \cong t_1$ and $t_2 \cong t_2$ hold. \square

Patterns Used in the Anti-Unify Module Refactoring

```
1 map f [] = []
2 map f (x:xs) = f x : map f xs
3
4 fold z f [] = z
5 fold z f (x:xs) = f x (fold z f xs)
6
7 foldl f z [] = z
8 foldl f z (x:xs) = foldl f (f z x) xs
9
10 foldlFlip f z [] = z
11 foldlFlip f z (x:xs) = foldlFlip f (f x z) xs
12
13 foldr2 f z ([],[]) = z
14 foldr2 f z ((x:xs),(y:ys)) = f x y (foldr2 z (xs,ys))
15
16 foldl2 f z ([],[]) = z
17 foldl2 f z ((x:xs),(y:ys)) =
18     foldl2 f (f z x y) (xs,ys)
19
20 data BTree a = L a | B (BTree a) (BTree a)
21
```

```
22 foldBTree f g (L a) = g a
23 foldBTree f g (B l r) =
24   f (foldBTree f g l) (foldBTree f g r)
25
26 data CTree a = E | C a (CTree a) (CTree a)
27
28 fold2CTree f g h z (E,E) = z
29 fold2CTree f g h z ((C x xl xr),E) =
30   h x (fold2CTree f g h z (xl,E))
31       (fold2CTree f g h z (xr,E))
32 fold2CTree f g h z (E,(C y yl yr)) =
33   g y (fold2CTree f g h z (E,yl))
34       (fold2CTree f g h z (E,yr))
35 fold2CTree f g h z ((C x xl xr),(C y yl yr)) =
36   f x y (fold2CTree f g h z (xl,yl))
37       (fold2CTree f g h z (xr,yr))
38
39 scanl f z [] = [z]
40 scanl f z (x:xs) = f x z : scanl f (f x z) xs
41
42 zip ([],[]) = []
43 zip ((x:xs),(y:ys)) = (x,y) : zip (xs,ys)
44
45 zipWith f ([],[]) = []
46 zipWith f ((x:xs),(y:ys)) =
47   f x y : (zipWith f (xs,ys))
```

Bibliography

- [1] Andreas Abel. 'Termination checking with types'. In: *ITA* 38.4 (2004), pp. 277–319.
- [2] Joonseon Ahn and Taisook Han. 'An Analytical Method for Parallelization of Recursive Functions'. In: *Parallel Processing Letters* 10.1 (2000), pp. 87–98.
- [3] Marco Aldinucci. 'Automatic program transformation: The Meta tool for skeleton-based languages'. In: *Constructive Methods for Parallel Programming, Advances in Computation: Theory and Practice* (2002), pp. 59–78.
- [4] Frances E. Allen and John Cocke. 'A Program Data Flow Analysis Procedure'. In: *Commun. ACM* 19.3 (1976), pp. 137–147.
- [5] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001. ISBN: 1-55860-286-0.
- [6] Corinne Ancourt and François Irigoin. 'Scanning Polyhedra with DO Loops'. In: *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, Williamsburg, Virginia, USA, April 21-24, 1991. 1991, pp. 39–50.
- [7] Russell R. Atkinson and Carl Hewitt. 'Parallelism and Synchronization in Actor Systems'. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, January 1977. 1977, pp. 267–280.

- [8] Emil Axelsson, Koen Claessen et al. 'The Design and Implementation of Feldspar - An Embedded Language for Digital Signal Processing'. In: *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*. 2010, pp. 121–136.
- [9] Adam D. Barwell, Christopher Brown and Kevin Hammond. 'Finding parallel functional pearls: automatic parallel recursion scheme detection in Haskell functions via anti-unification'. In: *Future Generation Computer Systems* In press (2017). issn: 0167-739X.
- [10] Adam D. Barwell, Christopher Brown et al. 'Towards Semi-automatic Data-type Translation for Parallelism in Erlang'. In: *Proceedings of the 15th International Workshop on Erlang, Nara, Japan, September 18-22, 2016*. 2016, pp. 60–61.
- [11] Adam D. Barwell, Christopher Brown et al. 'Using Program Shaping and Algorithmic Skeletons to Parallelise an Evolutionary Multi-Agent System in Erlang'. In: *Computing and Informatics* 35.4 (2016), pp. 792–818.
- [12] Adam D. Barwell and Kevin Hammond. 'In Search of a Map: Using Program Slicing to Discover Potential Parallelism in Recursive Functions'. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. FHPC 2017. Oxford, UK: ACM, 2017, pp. 30–41. isbn: 978-1-4503-5181-2.
- [13] Cédric Bastoul. 'Code Generation in the Polyhedral Model Is Easier Than You Think'. In: *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*. 2004, pp. 7–16.
- [14] Evgenij Belikov, Pantazis Deligiannis et al. *A Survey of High-Level Parallel Programming Models*. Technical Report. Heriot-Watt University, 2016.
- [15] Richard S. Bird and Lambert Meertens. 'Two Exercises Found In a Book on Algorithmics'. In: *Program Specification and Transformation*. Ed. by Lambert Meertens. North-Holland, 1987, pp. 451–457.

- [16] Guy E. Blelloch, Jonathan C. Hardwick et al. 'Implementation of a Portable Nested Data-Parallel Language'. In: *J. Parallel Distrib. Comput.* 21.1 (1994), pp. 4–14.
- [17] István Bozó, Viktoria Fordós et al. 'Discovering parallel pattern candidates in Erlang'. In: *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang, Gothenburg, Sweden, September 5, 2014*. 2014, pp. 13–23.
- [18] Christopher Mark Brown. 'Tool support for refactoring Haskell programs'. PhD thesis. University of Kent, UK, 2008.
- [19] Christopher Brown, Marco Danelutto et al. 'Cost-Directed Refactoring for Parallel Erlang Programs'. In: *International Journal of Parallel Programming* 42.4 (2014), pp. 564–582.
- [20] Christopher Brown, Kevin Hammond et al. 'Paraphrasing: Generating Parallel Programs Using Refactoring'. In: *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*. 2011, pp. 237–256.
- [21] Christopher Brown, Huiqing Li and Simon J. Thompson. 'An Expression Processor: A Case Study in Refactoring Haskell Programs'. In: *Trends in Functional Programming - 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers*. 2010, pp. 31–49.
- [22] Christopher Brown, Hans-Wolfgang Loidl and Kevin Hammond. 'ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques'. In: *Trends in Functional Programming, 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers*. 2011, pp. 82–97.
- [23] Christopher Brown and Simon J. Thompson. 'Clone detection and elimination for Haskell'. In: *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19, 2010*. 2010, pp. 111–120.

- [24] Peter E. Bulychev, Egor V. Kostylev and Vladimir A. Zakharov. ‘Anti-unification Algorithms and Their Applications in Program Analysis’. In: *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers.* 2009, pp. 413–423.
- [25] Peter Bulychev and Marius Minea. ‘An evaluation of duplicate code detection using anti-unification’. In: *Proc. 3rd International Workshop on Software Clones.* 2009.
- [26] Michael G. Burke and Ron Cytron. ‘Interprocedural dependence analysis and parallelization’. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986.* 1986, pp. 162–175.
- [27] Rod M. Burstall and John Darlington. ‘A Transformation System for Developing Recursive Programs’. In: *J. ACM* 24.1 (1977), pp. 44–67.
- [28] David R. Butenhof. *Programming with POSIX Threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-63392-2.
- [29] Colin Campbell and Ade Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. 1st. Microsoft Press, 2011. ISBN: 0735651752, 9780735651753.
- [30] David Castro. ‘Structured Arrows: a Type-Based Framework for Structured Parallelism’. PhD thesis. The University of St Andrews, 2017.
- [31] David Castro, Kevin Hammond and Susmit Sarkar. ‘Farms, pipes, streams and reforestation: reasoning about structured parallel processes using types and hylomorphisms’. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016.* 2016, pp. 4–17.
- [32] David Castro, Kevin Hammond et al. ‘Automatically deriving cost models for structured parallel processes using hylomorphisms’. In: *Future Generation Computer Systems* (2017).

- [33] Manuel M. T. Chakravarty, Gabriele Keller et al. 'Accelerating Haskell array codes with multicore GPUs'. In: *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*. 2011, pp. 3–14.
- [34] Barbara Chapman, Gabriele Jost and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN: 0262533022, 9780262533027.
- [35] Yun-Yan Chi and Shin-Cheng Mu. 'Constructing List Homomorphisms from Proofs'. In: *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*. 2011, pp. 74–88.
- [36] Wei-Ngan Chin, Akihiko Takano and Zhenjiang Hu. 'Parallelization via Context Preservation'. In: *Proceedings of the 1998 International Conference on Computer Languages, ICCL 1998, Chicago, IL, USA, May 14-16, 1998*. 1998, pp. 153–163.
- [37] Daniel H. Cohen. 'Conditionals, quantification, and strong mathematical induction'. In: *J. Philosophical Logic* 20.3 (1991), pp. 315–326.
- [38] Murray Cole. 'Algorithmic skeletons : a structured approach to the management of parallel computation'. PhD thesis. University of Edinburgh, UK, 1988.
- [39] Murray Cole. 'Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming'. In: *Parallel Computing* 30.3 (2004), pp. 389–406.
- [40] Gilberto Contreras and Margaret Martonosi. 'Characterizing and improving the performance of Intel Threading Building Blocks'. In: *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*. 2008, pp. 57–66.

- [41] Andrew Cook. ‘Using proof in transformation synthesis for automatic parallelisation’. PhD thesis. Heriot-Watt University, Edinburgh, UK, 2001.
- [42] Massimo Coppola and Marco Vanneschi. ‘High-performance data mining with skeleton-based structured parallel programming’. In: *Parallel Computing* 28.5 (2002), pp. 793–813.
- [43] Danny Dig. ‘A Refactoring Approach to Parallelism’. In: *IEEE Software* 28.1 (2011), pp. 17–22.
- [44] Gilles Dowek. ‘Higher-Order Unification and Matching’. In: *Handbook of Automated Reasoning (in 2 volumes)*. 2001, pp. 1009–1062.
- [45] Alfons Geser and Sergei Gorlatch. ‘Parallelizing functional programs by generalization’. In: *J. Funct. Program.* 9.6 (1999), pp. 649–673.
- [46] Jeremy Gibbons. ‘The third homomorphism theorem’. In: *Journal of Functional Programming* 6.4 (1996).
- [47] Jeremy Gibbons and Geraint Jones. ‘The Under-Appreciated Unfold’. In: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP ’98), Baltimore, Maryland, USA, September 27-29, 1998*. 1998, pp. 273–279.
- [48] Andrew John Gill, John Launchbury and Simon L. Peyton Jones. ‘A Short Cut to Deforestation’. In: *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*. 1993, pp. 223–232.
- [49] Andy Gill and Graham Hutton. ‘The worker/wrapper transformation’. In: *J. Funct. Program.* 19.2 (2009), pp. 227–251.
- [50] Horacio González-Vélez and Mario Leyton. ‘A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers’. In: *Softw., Pract. Exper.* 40.12 (2010), pp. 1135–1160.
- [51] Sergei Gorlatch. *Constructing list homomorphisms*. Technical Report MIP-9512. University of Passau, Germany, 1995.
- [52] Saul Gorn. ‘Letter to the Editor’. In: *Communications of the ACM* 1.1 (Jan. 1958), p. 1. issn: 0001-0782 (print), 1557-7317 (electronic).

- [53] Martin Griebel, Paul Feautrier and Christian Lengauer. ‘Index Set Splitting’. In: *International Journal of Parallel Programming* 28.6 (2000), pp. 607–631.
- [54] Jing Guo, Jeyarajan Thiayagalingam and Sven-Bodo Scholz. ‘Breaking the GPU programming barrier with the auto-parallelising SAC compiler’. In: *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*. 2011, pp. 15–24.
- [55] Geoff W. Hamilton. ‘Distillation: extracting the essence of programs’. In: *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*. 2007, pp. 61–70.
- [56] Kevin Hammond, Marco Aldinucci et al. ‘The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems’. In: *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*. 2011, pp. 218–236.
- [57] Kevin Hammond, Jost Berthold and Rita Loogen. ‘Automatic Skeletons in Template Haskell’. In: *Parallel Processing Letters* 13.3 (2003), pp. 413–424.
- [58] *Haskell Tools*. 2017. URL: <https://github.com/haskell-tools/haskell-tools>.
- [59] Ralf Hinze and Nicolas Wu. ‘Unifying structured recursion schemes - An Extended Study’. In: *J. Funct. Program.* 26 (2016), e1.
- [60] Steve Holzner. *Eclipse: A Java Developer’s Guide*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2004. ISBN: 0596006411.
- [61] Zhenjiang Hu, Hideya Iwasaki and Masato Takeichi. ‘Deriving Structural Hylomorphisms From Recursive Definitions’. In: *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP ’96), Philadelphia, Pennsylvania, May 24-26, 1996*. 1996, pp. 73–82.

- [62] Zhenjiang Hu, Hideya Iwasaki and Masato Takeichi. ‘Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms’. In: *ACM Trans. Program. Lang. Syst.* 19.3 (1997), pp. 444–461.
- [63] Zhenjiang Hu, Masato Takeichi and Wei-Ngan Chin. ‘Parallelization in Calculational Forms’. In: *POPL ’98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 1998, pp. 316–328.
- [64] Zhenjiang Hu, Masato Takeichi and Hideya Iwasaki. ‘Diffusion: Calculating Efficient Parallel Programs’. In: *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*. 1999, pp. 85–94.
- [65] Gérard P. Huet. ‘Higher Order Unification 30 Years Later’. In: *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*. 2002, pp. 3–12.
- [66] Gérard P. Huet. ‘The Undecidability of Unification in Third Order Logic’. In: *Information and Control* 22.3 (1973), pp. 257–267.
- [67] Gérard P. Huet. ‘The Zipper’. In: *J. Funct. Program.* 7.5 (1997), pp. 549–554.
- [68] Klaus Indermark and Herbert Klaeren. ‘Efficient Implementation of Structural Recursion’. In: *Fundamentals of Computation Theory, International Conference FCT’87, Kazan, USSR, June 22-26, 1987, Proceedings*. 1987, pp. 204–213.
- [69] Vladimir Janjic, Adam D. Barwell and Kevin Hammond. ‘Using Erlang Skeletons to Parallelise Realistic Medium-scale Parallel Programs’. In: *Proceedings of the Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems 2014*. 2014.

- [70] Vladimir Janjic, Christopher Brown and Kevin Hammond. ‘Lapedo: Hybrid Skeletons for Programming Heterogeneous Multicore Machines in Erlang’. In: *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK. 2015*, pp. 185–195.
- [71] Vladimir Janjic, Christopher Brown et al. ‘RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications’. In: *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016. 2016*, pp. 288–295.
- [72] Canghong Jin, Ze-min Li et al. ‘FastFlow: Efficient Scalable Model-Driven Framework for Processing Massive Mobile Stream Data’. In: *Mobile Information Systems 2015 (2015)*, 818307:1–818307:18.
- [73] Thomas Johnsson. ‘Lambda Lifting: Treansforming Programs to Recursive Equations’. In: *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings. 1985*, pp. 190–203.
- [74] Venkatesh Kannan. ‘Transformation of functional programs for identification of parallel skeletons’. PhD thesis. Dublin City University, 2017.
- [75] Ken Kennedy, Kathryn S. McKinley and Chau-Wen Tseng. ‘Interactive Parallel Programming using the ParaScope Editor’. In: *IEEE Trans. Parallel Distrib. Syst.* 2.3 (1991), pp. 329–341.
- [76] Tamás Kozsik, Melinda Tóth and István Bozó. ‘Free the Conqueror! Refactoring divide-and-conquer functions’. In: *Future Generation Computer Systems* (2017). issn: 0167-739X.
- [77] Leslie Lamport. ‘The Parallel Execution of DO Loops’. In: *Commun. ACM* 17.2 (1974), pp. 83–93.
- [78] John Launchbury and Tim Sheard. ‘Warm Fusion: Deriving Build-Cata’s from Recursive Definitions’. In: *Proceedings of the seventh international conference on Functional programming languages and*

- computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. 1995, pp. 314–323.
- [79] Daan Leijen, Wolfram Schulte and Sebastian Burckhardt. ‘The design of a task parallel library’. In: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. 2009, pp. 227–242.
- [80] Alan Leung, Ondrej Lhoták and Ghulam Lashari. ‘Automatic parallelization for graphics processing units’. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, Calgary, Alberta, Canada, August 27-28, 2009*. 2009, pp. 91–100.
- [81] Huiqing Li and Simon J. Thompson. ‘A Domain-Specific Language for Scripting Refactorings in Erlang’. In: *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 501–515.
- [82] Huiqing Li and Simon J. Thompson. ‘Safe Concurrency Introduction through Slicing’. In: *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015*. 2015, pp. 103–113.
- [83] Huiqing Li, Simon J. Thompson et al. ‘Refactoring with wrangler, updated: data and process refactorings, and integration with eclipse’. In: *Proceedings of the 7th ACM SIGPLAN workshop on ER-LANG, Victoria, BC, Canada, September 27, 2008*. 2008, pp. 61–72.
- [84] Yu Liu, Zhenjiang Hu and Kiminori Matsuzaki. ‘Towards Systematic Parallel Programming over MapReduce’. In: *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*. 2011, pp. 39–50.

- [85] Rita Loogen. ‘Eden - Parallel Functional Programming with Haskell’. In: *Central European Functional Programming School - 4th Summer School, CEFPS 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*. 2011, pp. 142–206.
- [86] Simon Marlow. ‘Parallel and Concurrent Programming in Haskell’. In: *Central European Functional Programming School - 4th Summer School, CEFPS 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*. 2011, pp. 339–401.
- [87] Simon Marlow, Ryan Newton and Simon L. Peyton Jones. ‘A monad for deterministic parallelism’. In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. 2011, pp. 71–82.
- [88] Motohiko Matsuda, Mitsuhisa Sato and Yutaka Ishikawa. ‘Parallel Array Class Implementation Using C++ STL Adaptors’. In: *Scientific Computing in Object-Oriented Parallel Environments, ISCOPE 97, Marina del Rey, CA, USA, December 8-11, 1997, Proceedings*. 1997, pp. 113–120.
- [89] Erik Meijer. ‘More Advice on Proving a Compiler Correct: Improve a Correct Compiler’. In: *Declarative Programming, Sasbachwalden 1991, PHOENIX Seminar and Workshop on Declarative Programming, Sasbachwalden, Black Forest, Germany, 18-22 November 1991*. 1991, pp. 255–273.
- [90] Erik Meijer, Maarten M. Fokkinga and Ross Paterson. ‘Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire’. In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. 1991, pp. 124–144.
- [91] Tom Mens and Tom Tourwé. ‘A Survey of Software Refactoring’. In: *IEEE Trans. Software Eng.* 30.2 (2004), pp. 126–139.
- [92] Akimasa Morihata. ‘A short cut to parallelization theorems’. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013*. 2013, pp. 245–256.

- [93] Akimasa Morihata, Kiminori Matsuzaki et al. 'The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer'. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 2009, pp. 177–185.
- [94] Kazutaka Morita, Akimasa Morihata et al. 'Automatic inversion generates divide-and-conquer parallel programs'. In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 2007, pp. 146–155.
- [95] Shin-Cheng Mu and Akimasa Morihata. 'Generalising and dualising the third list-homomorphism theorem: functional pearl'. In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 2011, pp. 385–391.
- [96] Kenji Nakayama. 'A new discrete Fourier transform algorithm using butterfly structure fast convolution'. In: *IEEE Trans. Acoustics, Speech, and Signal Processing* 33.5 (1985), pp. 1197–1208.
- [97] Mohan V. Nibhanupudi and Boleslaw K. Szymanski. 'Adaptive Parallelism in the Bulk-Synchronous Parallel Model'. In: *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II*. 1996, pp. 311–318.
- [98] William F. Opdyke. 'Refactoring Object-oriented Frameworks'. UMI Order No. GAX93-05645. PhD thesis. Champaign, IL, USA, 1992.
- [99] Will Partain. 'The nofib Benchmark Suite of Haskell Programs'. In: *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 July 1992*. 1992, pp. 195–202.
- [100] Lawrence C. Paulson. 'Mechanizing Coinduction and Corecursion in Higher-Order Logic'. In: *J. Log. Comput.* 7.2 (1997), pp. 175–204.

-
- [101] David A. Plaisted. ‘Handbook of Logic in Artificial Intelligence and Logic Programming (Vol. 1)’. In: ed. by Dov M. Gabbay, C. J. Hogger and J. A. Robinson. 1993. Chap. Equational Reasoning and Term Rewriting Systems, pp. 274–364.
 - [102] Gordon D. Plotkin. ‘A Note on Inductive generalization’. In: *Machine Intelligence* 5 (1970), pp. 153–163.
 - [103] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press books. Addison-Wesley, 1994. ISBN: 978-0-201-42294-8.
 - [104] *Programatica: Integrating Programming, Properties and Validation*. 2016. URL: <http://programatica.cs.pdx.edu>.
 - [105] John C Reynolds. ‘Transformational Systems and the Algebraic Structure of Atomic Formulas’. In: *Machine intelligence* 5.1 (1970), pp. 135–151.
 - [106] Martin P. Robillard. ‘Topology analysis of software dependencies’. In: *ACM Trans. Softw. Eng. Methodol.* 17.4 (2007), 18:1–18:36.
 - [107] Norman Scaife, Susumu Horiguchi et al. ‘A parallel SML compiler based on algorithmic skeletons’. In: *J. Funct. Program.* 15.4 (2005), pp. 615–650.
 - [108] T.F. Smith and M.S. Waterman. ‘Identification of common molecular subsequences’. In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. ISSN: 0022-2836.
 - [109] Georgios Tournavitis and Björn Franke. ‘Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information’. In: *19th International Conference on Parallel Architecture and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010*. 2010, pp. 377–388.
 - [110] José Manuel Calderón Trilla and Colin Runciman. ‘Improving implicit parallelism’. In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 2015, pp. 153–164.

- [111] Philip W. Trinder, Kevin Hammond et al. 'Algorithms + Strategy = Parallelism'. In: *J. Funct. Program.* 8.1 (1998), pp. 23–60.
- [112] Philip W. Trinder, Kevin Hammond et al. 'GUM: A Portable Parallel Implementation of Haskell'. In: *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania, May 21-24, 1996*. 1996, pp. 79–88.
- [113] Wojciech Turek. 'Erlang-based desynchronized urban traffic simulation for high-performance computing systems'. In: *Future Generation Computer Systems* (2017). ISSN: 0167-739X.
- [114] Abhishek Udupa, Kaushik Rajan and William Thies. 'ALTER: exploiting breakable dependences for parallelization'. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 2011, pp. 480–491.
- [115] Philip Wadler. 'A critique of Abelson and Sussman or why calculating is better than scheming'. In: *SIGPLAN Notices* 22.3 (1987), pp. 83–94.
- [116] Zheng Wang, Georgios Tournavitis et al. 'Integrating profile-driven parallelism detection and machine-learning-based mapping'. In: *TACO* 11.1 (2014), 2:1–2:26.
- [117] Mark Weiser. 'Program Slicing'. In: *IEEE Trans. Software Eng.* 10.4 (1984), pp. 352–357.